ROS2 Integration of ABB IRB 14000 YuMi

Marius Nilsen

The Department of Mechanical and Industrial Engineering (MTP)
Norwegian University of Science and Technology (NTNU)
December 11, 2019

Abstract

Small parts assembly puts requirements on the dexterity and maneuverability of the robot manipulator. To facilitate small part assembly, dual-arm robots like ABB IRB 14000 YuMi is developed. Featuring 7 joint on each arm, padding, and safety mechanisms for safe operation side-by-side human workers, this type of industrial robot presents a shift in industrial robotics. The new collaborative industrial robots introduce new possibilities for flexible robotics. This, however, requires new development strategies to be employed. When implemented in a dynamically changing environment, the ability to react to the environment in a timely manner also becomes of importance. Developing robot software is challenging, especially with real-time requirements. However, with the advent of ROS2 this is made easier. ROS2 featuring DDS and Quality of Service options enables real-time systems to be developed on ROS2. In this report, integration of the ABB IRB 14000 YuMi with ROS2 is presented. Using the Externally Guided Motion (EGM) interface from ABB, a ROS2 control system is able to control YuMi to track a continuously changing references in real-time, using a simple proportional gain joint position control.

Table of Contents

List of Tables

List of Figures

Chapter 1

Introduction

1.1 Problem description

Small parts assembly puts requirements on the dexterity and maneuverability of the robot manipulator. To facilitate small part assembly, dual-arm robots like ABB IRB 14000 YuMi is developed. Featuring 7 joint on each arm, padding, and safety mechanisms for safe operation side-by-side human workers, this type of industrial robot present a shift in industrial robotics. When implemented in a dynamically changing environment, the ability to react to the environment in a timely manner also becomes of importance. Developing robot software is challenging, especially with real-time requirements. However, with the advent of ROS2 this is made easier. The main objectives of this project are:

- Integration of the ABB IRB 14000 YuMi with ROS2
- Implement a ROS2 control system able to, in real-time, track continuously changing references.
- Evaluate the control systems real-time performance.

1.2 Structure

The report is organized into chapters. Chapter 2 reviews some preliminaries to form a common vocabulary. Chapter 3 reviews the robot in question, the ABB IRB 14000 YuMi, and presents the ROS2 integration of YuMi. Further, in chapter 4, a discussion is had about how to evaluate the performance of the developed ROS2 control system, key performance indicators are defined, and two tests are presented. Results are shown in chapter 5 and discussed in chapter 6. The report is then concluded in chapter 7, and rounded of with further work.

Chapter 2

Preliminaries

Before embarking on the robot system and the ROS2 integration of YuMi, some necessary preliminaries will be reviewed.

2.1 Manipulator Kinematics

To be able to control a robot and program its motion, we are in need of a formalism to describe its motion. Kinematics is a branch of classical mechanics that describes the geometric representation of motion and is often used to describe multi-link systems, e.g., robot manipulators, motions. The section begins with a brief presentation of some important concepts from kinematics, following this a short review of how robots can be modeled using kinematics and lastly brief presentations of forward kinematics, inverse kinematics, and redundant manipulators. This section is based on information found in [1] and [2].

Kinematics describes the motion of points, geometric objects, and groups of geometric objects without considering the forces that cause them to move. Kinematic problems are typically solved using geometric relationships and known initial conditions to derive the conditions of the unknown points in the system.

Representing position

In Euclidean geometry, a point is defined as a geometric object without width, length, depth, or volume. A common interpretation is that a point captures a unique location in Euclidean space. In kinematics, a point is used to represent a position. Given a known coordinate frame, a position of a point can be represented by a vector p from the origin of the coordinate frame to the location of the point.

Rigid bodies

A solid body where deformation is zero or negligible is considered a rigid body. The distance between any two points on a rigid body remains constant over time regardless of forces exerted on it. Kinematics is primarily concerned with rigid bodies. The position of a rigid body is described by the position of its particles. Having zero deformation, the particles of the rigid body have constant a relation. Hence, the position of a rigid body can be derived from knowing the position of three non-colinear particles. A rigid body is completely described in three-dimensional space by its pose in relation to a reference coordinate frame. A pose is composed of an object's orientation and position in reference to a reference coordinate frame. Both position and orientation provide three degrees of freedom, giving a rigid body six degrees of freedom in total. However, a rigid body is normally described in three-dimensional space by its position and orientation (*pose*) with respect to a reference frame. In three dimensional space, this means the body has three rotational and three positional degrees of freedom, giving it a total of six.

Representing orientation

The orientation of a geometric object encompasses how it is placed in the space it occupies. Orientations of rigid bodies are generally described by attaching a coordinate frame to the rigid body and describing the orientation of the coordinate frame with reference to a fixed, predefined coordinate frame outside the body. The orientation of the attached coordinate frame with respect to the fixed reference coordinate frame is usually described by rotational matrices.

Representing translation

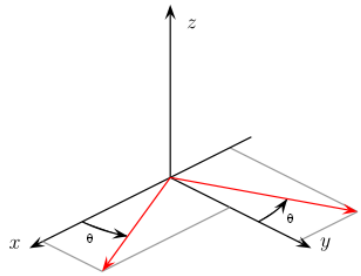
A movement of a rigid body not rotating its coordinate frame with respect to a fixed reference frame is denoted as translation. A translation has similar to position, three degrees of freedom.

The Rotation Matrix

A rotation matrix is a 3x3 matrix with three degrees of freedom. A rotation of a coordinate frame A with respect to a coordinate frame B is noted as R_B^A. The orientation of the unit vectors of the rotated coordinate frame with reference to the original coordinate frame gives the columns of the rotation matrix, with column one being the orientation of the unit vector of the x-axis, column two the orientation of the unit vector of the y-axis and finally column three the orientation of the unit vector of the z-axis. The simplest three-dimensional rotation matrix represents a counter-clockwise rotation θ about one of the axes of the original coordinate frame. A rotation θ about the z-axis is denoted $R_z(\theta)$. The

rotation matrix of the rotation is listed in ... and illustrated in ..

$$R_z(\theta) = \begin{bmatrix} \cos\theta & -\sin\theta & 0 \\ \sin\theta & \cos\theta & 0 \\ 0 & 0 & 1 \end{bmatrix}$$



The rotation matrix is a member of the rotation group $SO(3)$, which is the 3x3 subgroup in the special orthogonal group $SO(n)$. The $SO(3$ group encompasses square matrices with real entries, orthogonal columns, and determinant 1. Formalized this reads as:

$$SO(3) = \left\{ \boldsymbol{R} | \boldsymbol{R} \in \mathbb{R}^{3\times 3}, \quad \boldsymbol{R}^{\mathrm{T}}\boldsymbol{R} = \boldsymbol{I} \quad \text{and} \quad \det \boldsymbol{R} = 1 \right\} \tag{2.1}$$

The Homogeneous Transformation Matrix

The description of a rigid body's change of position, translation, and orientation is denoted as its homogeneous transformation and represented using a homogeneous transformation matrix. It provides a compact notation for expressing the coordinate transformation between two frames. In fig. 2.1, a transformation between a frame a and frame b illustrated.

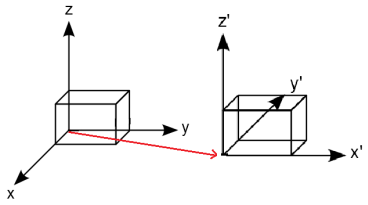


Figure 2.1: A transformation containing a translation and rotation.

The subscript and superscript notation is the same as with rotational matrices, a transformation from a coordinate frame a to a coordinate frame b is noted as R_b^a. The homogeneous transformation has dimensions 4x4 and is often given in the form listed below in **??** In this notation the rotational component of the transformation is given by R_b^a and the translational component by r_{ab}^a. R_b^a represent the rotation from frame a to b, and r_{ab}^a

represent the translation from frame a to b given in the coordinates of a.

$$\boldsymbol{T}_b^a = \begin{bmatrix} \boldsymbol{R}_b^a & \boldsymbol{r}_{ab}^a \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} \tag{2.2}$$

The transformation matrix is a member of the Special Euclidean Group $SE(3)$:

$$SE(3) = \left\{ \boldsymbol{T} | \boldsymbol{T} = \begin{bmatrix} \boldsymbol{R} & \boldsymbol{r} \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix}, \boldsymbol{R} \in SO(3), \boldsymbol{r} \in \mathbb{R}^3 \right\} \tag{2.3}$$

The inverse of a homogeneous transformation $\boldsymbol{T}_b^a$ is the same as the transformation back to the original pose:

$$(\boldsymbol{T}_b^a)^{-1} = \begin{bmatrix} (\boldsymbol{R}_b^a)^{\mathrm{T}} & -(\boldsymbol{R}_b^a)^{\mathrm{T}} \boldsymbol{r}_{ab}^a \\ 0^{\mathrm{T}} & 1 \end{bmatrix} = \begin{bmatrix} \boldsymbol{R}_a^b & \boldsymbol{r}_{ba}^b \\ \mathbf{0}^{\mathrm{T}} & 1 \end{bmatrix} = \boldsymbol{T}_a^b \tag{2.4}$$

Additionally it can be shown that the transformation from a frame a to a frame c can be given as the composite transformation:

$$T_c^a = T_b^a T_c^b \tag{2.5}$$

Modelling of Robot Manipulators

Robot manipulators are modeled using links and joints, forming either open or closed kinematic chains. An open kinematic chain features a free endpoint, whereas a closed chain does not. A robot arm is a typical open kinematic chain. When modeling robot manipulators, the links of the robot are modeled as rigid bodies, and the joints are assumed to provide either pure rotation or pure translation. Meaning all joints can be categorized into two categories: pure rotational (revolute) joints and pure transnational (prismatic) joints. Typically graphical representations of the two joint types are shown in fig. 2.2 below.

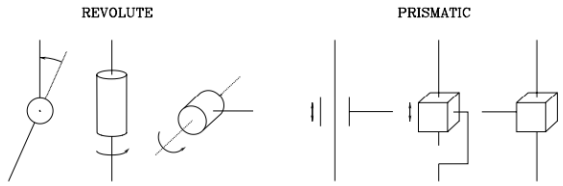


Figure 2.2: There are several representation of the two joint types. Figure from [1]

Joints and links are put together to represent a manipulator. Shown below in fig. 2.3

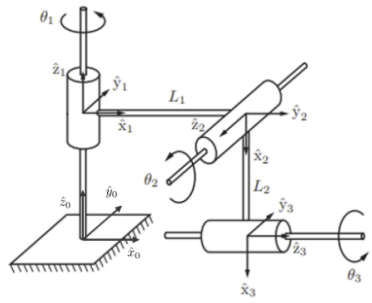


Figure 2.3: A modelled three dof robot manipulator. Figure adapted from [2]

Denavit-Hartenberg parameters

Each joint in a manipulator chain is attached with a coordinate frame. The transformations between the coordinate frames are represented using homogeneous transformations. Attaching reference frames to joints can be done using the Denavit-Hartenberg convention [3]. The Denavit-Hartenberg convention uses four Denavit-Hartenberg (DH) parameters and a specific convention for detailing how the reference systems are placed and oriented.

Forward Kinematics

Forward kinematics is the practice of using joint configurations and kinematics to determine the pose of the end-effector. This is typically done through chains of homogeneous transformations. Homogeneous transformations are composable, meaning a transformation from frame 0 to frame n be found by either transforming from joint to joint, or as a direct transformation from frame 0 to frame n:

$$\boldsymbol{T}_n^0 = \boldsymbol{T}_1^0 \boldsymbol{T}_2^1 \dots \boldsymbol{T}_n^{n-1} \tag{2.6}$$

$$\boldsymbol{T}_e^0 = \begin{bmatrix} \boldsymbol{R}_e^0 & \boldsymbol{p}_{0e}^0 \\ \boldsymbol{0}^{\mathrm{T}} & 1 \end{bmatrix} \tag{2.7}$$

Inverse Kinematics

Inverse kinematics is concerned with the opposite task, finding the joint configuration giving the desired pose of the end-effector. For a 6 degrees-of-freedom (dof) manipulator, a closed-form analytical solution can be found for the desired pose. However, with more than 6 dof a and no other conditions applied, a closed-form solution may not be found. Besides analytical solutions, there are also iterative and other numerical methods.

Redundant manipulators

Redundant manipulators are all manipulators with more than 6 joints. A typical redundant manipulator has seven joints, meaning it got an extra degree of freedom. This extra degree of freedom allows it to move while holding the end-effector at specific pose, or otherwise change its configuration while holding its end-effector in a fixed position and orientation.

Joint Space, Cartesian Space and Singularities

Throughout the report, three more concepts will be used: Joint Space, Cartesian Space, and singularity. Joint space denotes the space in which the configurations of the robot's joints can be easily defined. The joint space allows the position of the robot to be defined by the value of its axis. Cartesian space is the three-dimensional space in which the robot resides. In Cartesian space, positions are given as three dimensional (x, y, z) vector. A singularity is a condition where a robot manipulator loses one or more degree of freedom, and where a change in a joint variable may not result in a change of end-effector pose.

2.2 Control Systems

The integration of YuMi in ROS2 will include a control system to be developed. Therefore some key concepts from the field control theory will be reviewed. This section will be scoped to Linear time-invariant (LTI) single-input-single-output (SISO) control systems only. The section is based on information from [4] and [5].

A control system is a system commanding and managing an entities behavior. A control system minimally consists of an entity, or process, to be controlled and a controller managing the process's behavior, or response. The controller takes the desired output, or setpoint, for the process as input and formulates an input to the system based on selected update law. An update law, or control law, is a predefined mathematical function for converting the controller's input to a system input. The input to the system is the entity used to modify the process's behavior. This can be any real physical quantity the physical controller is able to produce.

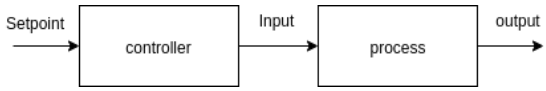


Figure 2.4: Entities of a minimal control system.

Control theory, the underlying theory of control systems, is the theory of determining an input to receive the desired output, or response. Control theory is a broad field, and as mentioned in the introduction to the section, this review will be scoped to only consider

Linear time-invariant (LTI), single-input-single-output (SISO) control systems. For a system to be a linear system, two requirements must be satisfied: homogeneity and additivity. The homogeneity requirement tells that when multiplying the input by a constant, the response is multiplied with the same constant, i.e., If an input is doubled, the response is also doubled. The additivity requirement tells that the response from an input comprised of the sum of two other inputs is the sum of the responses to each of the two inputs presented separately. Together these two requirements are called the superposition principle. A time-invariant system is a system whose output is independent of time, meaning the same input always produces the same output.

Open- and Closed-loop Control Systems

Control systems often feature feedback. In control theory, feedback is the concept of feeding the response of the process backward to be used in earlier parts of the control system. The concept of feeding the response of the process backward to be used as an additional input to the controller is called feedback control. Feedback enables the control systems to monitor the process's response and is one of the most central concepts of control theory. Control systems are often categorized into two categories: open-loop systems not using feedback and closed-loop systems that do.

Open-loop Control Systems

The minimal control system presented earlier in fig. 2.4 is an open-loop control system. Open-loop control systems are control systems not utilizing feedback. Being unable to monitor the output of the process, open-loop control systems are often simple systems.

Closed-loop Control Systems

Closed-loop control systems are control systems using feedback control. Closed-loop control systems feed back a measurement of the process's output, which is used to compare with the desired output, the set point. In closed-loop control systems, the controller often takes the deviation between the desired output and the measured output, the error, as its input instead of simply the desired value. Thus, in the block diagram, forming a loop, as shown in fig. 2.5.

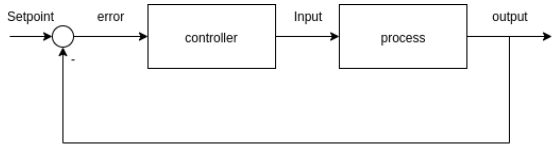


Figure 2.5: Entities of a minimal control system.

The P and PID Controller

The controller generates system input using an update law to calculate the input that produces the desired output from the controlled process. In the developed ROS2 architecture,

P and PID controllers are used and discussed. The following is, therefore, a brief presentation of the two controllers.

The P-controller
The proportional gain controller, or simply the P-controller, is one of the simplest controllers. The input to the process is proportional to the deviation between the desired value and the measured value. The update law is shown in eq. (2.8), $u(t)$ denotes the computes process input. The P-controller can never completely remove a steady-state error but will, however, reduce it. The P-controller is therefore unsuitable for systems requiring the exact value of the setpoint to be reached. On the contrary, many systems allow a small deviation to exists.

$$u(t) = K_{\mathrm{p}} e(t) \tag{2.8}$$

The PID-controller
A very popular controller is the proportional-integral-derivative (PID) controller. In addition to the proportional gain term, it utilizes two more terms: the integral-term, $K_{\mathrm{i}} \int_0^t e\left(t'\right) dt'$, and the derivative-term, $K_{\mathrm{d}} \frac{de(t)}{dt}$. Its update law is shown in eq. (2.9).

$$u(t) = K_{\mathrm{p}} e(t) + K_{\mathrm{i}} \int_0^t e\left(t'\right) dt' + K_{\mathrm{d}} \frac{de(t)}{dt} \tag{2.9}$$

The integral term can be tuned to remove the steady-state error the P-controller was unable to eliminate. The integral term continuously sums the error, providing a contribution to the process input that is proportional to the magnitude and duration of the error. The derivative term takes the derivative of the error curve, allowing the controller to anticipate changes in the response of the system. Its contribution to the process input is proportional to this derivative. A block diagram of a simple PID controller is shown in fig. 2.6

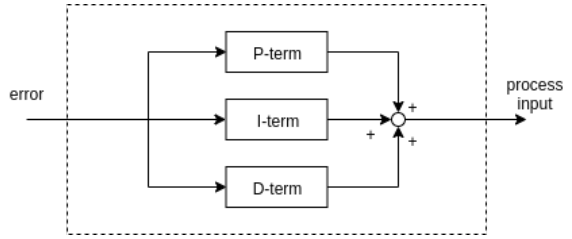


Figure 2.6: A simple PID controller.

2.3 Robot Operating System

Robot Operating System (ROS) [6] is a flexible framework for writing robot software. It consists of a collection of software packages, libraries, tools, and conventions useful for

developing software for robots. Its first iteration was released by Willow Garage in 2007, and its latest long term support release, Melodic Morenia, was released in 2018. ROS versions are released as distributions. Its distribution naming is alphabetic and normally composed of an adjective and a type of turtle of the version letter [7]. Meaning Kinetic Kame predates Lunar Loggerhead, which again predates Melodic Morenia.

While ROS is not an operating system in the traditional sense, it provides most of the services expected from an operating system, like hardware abstraction, package management, message-passing between processes, and implementations of commonly used functionalities. ROS is sometimes presented as the middleware or the underlying plumbing behind the processes and message-passing, and while that is true, ROS also delivers a wide range of tools and robot agnostic capabilities. ROS is open-source, and most ROS packages are open-source as well. ROS actively encourages collaborative software development.

ROS 2 was announced in 2015, with its first alpha released the same year. Its first real distribution, "Ardent Apalone", was released in 2017. Several significant changes were planned, which meant a complete revamp and thus breaking of APIs. Therefore, to conserve ROS1, ROS2 was from the start planned as a separate entity. ROS2's release schedule is at the moment a new distribution every 6 months, latest being Dashing Diademata, released May 2019. As the changes from original ROS1 to ROS2 changed large parts of the ROS infrastructure, ROS1 and ROS2 will be separately reviewed. The information in the section is based on the information found in ROS wikis [8].

2.3.1 ROS1

Before reviewing the ROS framework and its capabilities, we will review the components comprising a ROS system.

Nodes
Robotics software often comprises of many processes and applications communicating with each other, often as a distributed system across multiple machines. ROS is acting as a middleware and is responsible for the communication between the computational processes, in ROS called nodes. A node is an instance of a executable and may contain several threads. A node is usually a small independent program responsible for a single aspect or function of a system or device.

Nodelet
Nodelets are a specific type of node designed to run multiple nodes as threads in a single process. When executing code in the processor, switching between threads is significantly faster than a context switch between processes. Nodelets, therefore, offer greater performance for communication between performance-critical nodes. Nodelets are compiled to a shared library

Topics
Topics are named busses in which nodes exchange messages. Topics are asynchronous, and support many-to-many communication with anonymous publish/subscribe semantics.

Nodes that are interested in a type of data subscribe to the topic carrying that data, nodes that produce data publish to the relevant topic. The nodes don't know who, if any, are consuming or producing data. Topics are intended for unidirectional streaming of data and are strongly typed by the ROS message type used to create it. Topics are typically used for continuous data streams like sensor data or robot state.

Services
Services are another method for nodes to exchange data. Services provide synchronous communication between nodes in scenarios where one-way transport is not desirable, and a request/reply interaction is needed. Essentially, services are remote procedure calls, where one node calls a function that execute in another node. Service communication is similar to client/server communication. The server (node providing the service) specifies a callback to deal with the service request and advertise the service, the client (node utilizing the service) access this service through a local proxy. Service messages (.srv) are composed of a pair of messages, one for the request and one for the reply. Services are typically used for operations that are performed only occasionally, and where the execution time is short, or at least bounded. The data transfer is done without use of topics.

Actions
The last of the three core channels over which ROS nodes communicate is actions. Actions utilises a somewhat similar client/server format and request/response semantic to services. An action utilizes a goal to initiate as operation and receives a result when it is completed. However, an action offers feedback to provide updates on the progress of the operation. Goals can also be canceled. Services being synchronous means the node calling the service is blocked until it receives a request, leaving services unsuitable for unbounded or long-running tasks. Besides being preemptible, actions are asynchronous, meaning the node calling the cation can continue its execution while the action server is executing the action call. Actions are typically used for discrete, relatively long-running tasks like moving the robot. The data transfer of actions is implemented using topics. Essentially, actions are a higher-level protocol describing how a set of topics (goal, cancel, feedback, result) should be used in combination.

Master
While communication between nodes in ROS is peer-to-peer, discovery is not. This is handled by a centralized discovery service called the master. The master provides declaration and registration services to nodes in the system and enables nodes to locate one another. Communication between nodes is typically done via the TCP/IP protocol, or ROS' own TCPROS protocol. In addition, the master tracks publishers and subscribers to topics and also host a parameter server. The parameter server is a centralized, shared multivariate dictionary accessible to all nodes, and are used by nodes to store and retrieve data during runtime.

Naming and namespaces
To provide encapsulating, and avoid name collision, there are specific rules for naming in ROS. Nodes, topics, services, and parameters are collectively referred to as graph re-

sources, and every graph resource is identified by a short string called the graph resource name. There are four types of graph resource names in ROS: base, relative, global, and private. Base names are the name given to the node. Relative names are comprised of a namespace and a base name, separated by the forward slash (/). A namespace is a naming group similar to namespaces in C and C++. As node names are resolved relatively, nodes do not need to be aware of what namespace in which they are located. By adding a leading forward slash to a node name, it is fully resolved and global. Names with a leading " " are private. Private names use the name of their node as a namespace. The naming convention is illustrated for a node bar, in namespace foo with private name $priv$ below.

$$\text{/foo/bar/priv} \quad : \text{/foo} \quad \text{/bar} \quad \text{/priv} \tag{2.10}$$

$$\tag{2.11}$$

Computational graph
The computation graph is the peer-to-peer network of the nodes processing data together. The computational graph shows how the nodes interact with each other. The basic computation graph concepts of ROS are nodes, the master, the parameter server, messages, services, topics, and bags.

File system
ROS utilizes a file system whose main components are packages, metapackages, and workspaces. Packages are the most fundamental unit for organizing software in ROS. A package might contain nodes, datasets, configuration files, external libraries, or anything else which, when organized together, constitute a useful module. Packages are the most atomic unit in the ROS filesystem and contain the minimum content needed to create a ROS program. The goal of the ROS package is to organize software in a reusable and easy-to-consume manner. A ROS package should provide enough functionality to be useful while being kept small enough not to be heavyweight. Usually, a package is constrained to contain one independent unit, with one purpose.

Related packages can be grouped and represented by metapackages. Metapackages do not install any files and are used to simply aggregate packages into groups.

Finally, the directory containing a project's packages and metapackages is called a workspace. Packages in the same workspace can be built at the same time using the ROS build system. Nodes and packages can be utilized across workspaces, given they are built beforehand.

Summary
The ROS architecture has three levels of concepts: the filesystem level, computation graph level, and the community level. We have reviewed these levels from highest to lowest. We began reviewing the distribution system and release the schedule of ROS. Further, we reviewed the components in the computation graph level. Lastly, we reviewed the filesystem used in ROS.

Before reviewing the changes from ROS1 to ROS2, we will first review some key con-

cepts from real-time systems and network communication. The section on real_time systems is based on information from [9]. The section network communication is based on information from **??** and **??**.

2.4 Real-time systems

A real-time system is a system whose results correctness depends not only on the result but also on the time the result was delivered. Typically a real-time system is subject to external input stimuli and is often involved in systems interacting with the physical world. Typically are safety-critical systems like anti-lock brakes or the flaps on an airplane controlled by real-time systems. However, many robot systems also operate with real-time requirements.

2.4.1 Real-time computing

Real-time computing is computing with a specified time limit, or deadline, for its execution. For real-time systems to be able to deliver within its deadlines, they are concerned with guaranteed and predictable execution times on its tasks. A task which occasionally takes five times longer than its mean to execute renders the upper bound on the execution time also five times longer. As real-time systems are concerned with managing deadlines, the worst-case execution time of a task is, therefore, of greater interest than the mean. A real-time system must be able to guarantee that a task can be completed before its deadline.

2.4.2 Classification of Real-Time Systems

Real-time systems are typically classified by the consequence of missing a deadline. We operate with three categories of real-time systems:

Hard	Missing a single deadline is defined as a total system failure.
Firm	Infrequent missing of deadlines is tolerable, but degrades the systems performance. Results arriving after their deadlines are useless.
Soft	Missing a deadline is tolerable, but degrades the systems performance. The usefulness of a result degrades after its deadline.

Table 2.1: Classification of real-time systems.

Not all literature agree upon the existence of the *Firm* hardness level, and operate instead with only the *soft* and *hard* hardness levels. Regardless, in this work, we will operate with

all three. Additionally, continuing on-wards, real-time will refer to *hard* real-time.

2.4.3 Characteristics of Real-Time Systems

Manageable and predictable execution time on the tasks of a system is however not enough for the system to be a real-time system. The code itself is far from the only factor affecting the task's execution time. Therefore, a real-time system is from the ground up built to be able to deliver in accordance with its deadlines. A real-time system requires both an real-time safe operating system and a deterministic scheduling. Additionally, the real-world performance of a real-time system is not only comprised of the system's ability to deliver correct and timely responses, but also on the systems predictability and dependability.

A completely predictable system is a system which can be predicted perfectly in the future, knowing only the system's current state. For a fully predictable system, it is possible to prove, and thus guarantee, weather a deadline will be met. A perfectly predictable system is very similar to a deterministic system. Determinism implies a complete lack of randomness, meaning a deterministic system always produces the same output from the same set of starting conditions. There exists no fully predictable or deterministic systems in the real-world. Dependability is a measure of a systems availability, reliability and maintainability. Unexpected failures directly violate the principles of predictability and determinism, and must be avoided. Therefore do real-time systems often feature significant measures taken to handle fault, errs or module failures.

2.5 Network Communication

A large proportion of the changes between ROS1 and ROS2 are concerned with how data is transferred between nodes, and some new concepts and technologies are introduced. Therefore before we review ROS2, we will review some important concepts from network commendation.

2.5.1 The OSI-model

To establish a common vocabulary for this section and later sections, we will begin with a brief presentation of the OSI-model. The OSI-model was designed to standardize network architecture and promote interoperability [10]. The OSI-model describes the data flow through the computer from application to the physical medium of transportation, dividing the services, and handling of the data into seven layers:

1. Physical layer: Physical means of transmission of data
2. Data link layer: Transfer of bits over physical medium
3. Network layer: Handling of data addressing and routing
4. Transport Layer: Transport protocol
5. Session Layer: Handling of sessions between network nodes

6. Presentation layer: Data translation
7. Application layer: Methods for applications to interact with the network

2.5.2 Quality of Service

Quality of Service (QoS) is a network traffic control mechanism that manages data traffic to either differentiate between high priority traffic and traffic of lesser importance or guarantee certain transport conditions to some traffic flows. Conditions of concern are typically bandwidth. Latency, jitter, and packet loss. QoS is, therefore, often a valuable tool for achieving deterministic transport over a network.

2.5.3 Data Distribution Service

A data distribution service (DDS) is a data-centric end-to-end middleware managed by the Object Management Group (OMG). DDS utilizes a data-centric transport similar to ROS, and also similar to ROS, it integrates applications to function together as one. However, DDS employs a virtual global data space. To the applications, the global data space looks like native memory, and applications write and read to the data space via an API as if it was its local storage. In reality, the DDS sends messages to update the appropriate store on the remote nodes. Effectively ensuring operations are performed on the data directly instead of copies or reconstructions of the data. DDS addresses the needs of applications that require real-time data exchange, is decentralized, provides dynamic discovery, and can be configured with QoS specifications.

2.6 Robot Operating System 2

In this section, Robot Operating System 2 (ROS2) will be reviewed. As ROS1 already has been presented, the focus in this section will be on changes from ROS1 and ROS2.

Real-time ready

One of the primary goals for ROS2 was support for real-time systems. Due to the TCP based communication, dynamic memory allocation, and numerous other factors, ROS1 did not in itself, without any external framework, support real-time computation. Therefore ROS2 was written with this goal in mind, and as a consequence of the changes to be review below, it is possible to write real-time nodes in ROS2. To achieve a real-time system, it is, however, still necessary to run ROS2 in a real-time able operating system, and surround the code with a real-time able environment.

DDS as new middleware

ROS1 uses TCP as its transport protocol almost exclusively, and while having support for UDP transport, this transport is unreliable for big data transmissions and not uniformly supported. Relying heavily on TCP makes ROS1 unsuitable for communication over lossy networks, i.e., wireless. In addition, the TCP protocol is unsuitable for real-time applications, as retransmissions can lead to large delays and unpredictable transmission times. DDS offers both reliable and unreliable communication in addition to graceful degradation. DDS uses UDP for its transport of both reliable and unreliable data.

Quality of Service profiles introduced

Another change is the introduction of Quality of Service profiles for connections in the computational graph. It is possible in ROS2 to tune the required QoS of its connections. QoS settings are represented in ROS2 as QoS profiles. A QoS profile is comprised of a set of QoS policies that describes a specific requirement to an aspect of the connection. There are four available types of QoS policies in ROS 2:

- History - Message queue mode
- Depth - Message queue size
- Reliability - Delivery guarantee of messages
- Durability - Persistence of messages

Changes to the Computational Graph

Changes have also been made in elements of the computational graph. We will review some of the most significant.

No Master
With the change of middle ware to DDS, the master were no longer necessary in ROS2, and were therefore removed.

Components
Components are a new entity in ROS2 replacing nodelets. Similar to nodelets, components are compiled into a shared library, which can be loaded into a separate process or into a process containing other components. Loading several nodes into a process was a programming time decision in ROS1 as the nodelets used a different API from nodes. However, the component-based interface to nodes in ROS2 allows for components to be loaded and unloaded into processes at both compile-time and deploy time.

Managed Lifecycle
Besides using the component-based interface for nodes, nodes are expected to have a standardized and managed lifecycle. The primary goal of the lifecycle is to provide better control of the state of the ROS system by providing a known interface and execution by a known life cycle state machine. It allows, for instance, the launcher to ensure all compo-

nents are initialized correctly before allowing any components to execute, and for nodes to be restarted or replaced in a running system. The life cycle state machine can be seen in section 7.4.

Actions and Services
In ROS1, actions were implemented using a set of topics namespaced after the actions name, meaning it was generated a topic for each of the messages, a total of five topics. It was not possible to use services for any of the messages as services in ROS1 are synchronous. However, in ROS2, services are changed to be asynchronous, and as a result, ROS2 actions use a combination of topics and services. The distribution of action messages in services and topics are illustrated below.

Figure 2.7: Illustration showing how action messages are sent using a combination of topics and services. Illustration courtesy of ROS2 wiki

2.7 Relevant Web Technologies

Lastly, before transitioning to the system description, some relevant concepts within the field of web services will be reviewed. As some functionalities of the control software developed rely on this type of communication, it will be useful to conduct a brief review of some key terms and concepts.

HTTP
HTTP (HyperText Transfer Protocol) is an application layer protocol for request-response, client-server communication, and is the most common application layer protocol used on the internet. HTTP communication is half-duplex and mainly done through a client sending a request to a server responding with a status code. There a specified and limited set of possible client actions, and likewise a limited set of server status codes. Typical HTTP communication is based on URLs and verbs. The verbs represent the actions that can be performed, and the URLs defines were to perform the action. The URL's typically identify a web address. HTTP is stateless and connectionless, meaning the client and server only know about each other during the current request, and only connected during the transmis-

sion of requests and responses. Between transmissions, the client and server do not know about each other, nor are they connected.

Web Services
A web service is a computational entity accessible over the web via a standard web protocol, typically HTTP. Unlike web applications which interact with a user, web services are used by applications to communicate with other applications over the web. Web services are fully platform agnostic and communicate using a predefined data format. Webs services are therefor useful for decoupling and allows for entities from different platforms and programming languages to communicate with each other through a shared vocabulary.

REST
REST (Representational State Transfer) is an architecture style that defines a set of constraints for web services. For an entity to be RESTful it must comply with the REST rules for that entity. For instance, a RESTful API is an API that follows the API rules of the REST specification. Essentially, REST is a set of rules specifying how to use the HTTP protocol.

WebSockets
WebSockets is an application layer protocol that can be used as an alternative to HTTP. It allows for bi-directional (Full-duplex) communication over a single TCP connection between a client, typically a browser, and a server. WebSockets is, in contrast to HTTP, not connectionless. The connection between the client and the server is kept also after transmissions have ceased. Also, in contrast to HTTP, WebSockets facilitates event-driven responses without having to poll the server for a reply.

Google Protocol Bufffers
Google Protocol Buffers (protobuf) is a protocol for simple and efficient serialization and de-serialization of data [11]. Protobuf messages structures are described in .proto, proto definition files, and is later compiled. This definition file is made once and used by both the sender and receiver, which uses it to serialize and de-serialize the message in the manner defined in the definition file. An application receiving a protobuf message over the network first reads the serialized message from the network, de-serializes it, and reads the data from the de-serialized message. To send a protobuf message, an application likewise serializes it before sending it over the network. Protobuf is language-neutral and typically 10-100 times faster than XML [11]. Serialization converts messages to a binary format, and a noted disadvantage of protobuf is the increased difficulty of debugging network package[12].

Chapter 3

System description

3.1 ABB IRB 14000 ”YuMi”

The robot to be controlled is the ABB IRB 14000 ”YuMi.” YuMi is a dual-arm collaborative industrial assembly robot introduced at the Hannover Messe in 2015 by ABB [13]. It was designed to meet the production needs of the consumer electronics industry [14], where being safe to operate in collaboration with humans, being capable of precise and fast movements and the ability to operate in a confined space while maintaining a human-like reach were some the requirements identified by ABB [14], [15].

Figure 3.1: YuMi front facing equipped with the Smart Grippers

Being a collaborative robot, YuMi is designed to work alongside humans without any fencing or other safety. This requires collisions between YuMi’s arms and the environment to be non-harmful. As a consequence, the arms of YuMi are both lightweight and padded. Additionally, YuMi’s paddings are installed with force sensors and the robot controller with safety features stopping the motors upon detected collision. YuMi is very movable, weighting only 38 kgs [16], and is designed to facilitate effortless deployment. It can be powered by a normal wall socket and is easily fastened to a table by its table mountings

and a couple of bolts.

3.1.1 Technical data

The arms are identical and contain seven revolute joints, giving each arm seven degrees of freedom. By having an extra joint, YuMi is able to reach a much greater range of poses while maintaining a large workspace. The workspace is shown in fig. 3.2. Some useful technical data is presented in table 3.1

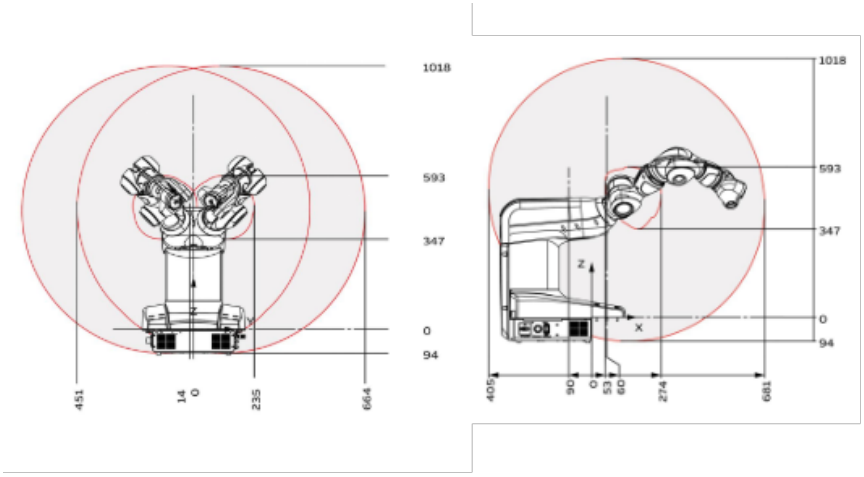


Figure 3.2: YuMi's workspace shown from the front and side.

Property	Value
Degrees of freedom	7 per arm
Payload	0.5 kg
Reach	559 mm
Width, Length, height [mm]	399*497*571
Weight	38 kg
Max TCP Velocity	1-5 m/s
Max TCP Acceleration	11 m/s*s
Acceleration time 0-1m/s	0.12s
Position repeatability	0.02 mm

Table 3.1: YuMi Specifications, data from [16].

3.1.2 Embedded IRC5 robot controller

YuMi features an embedded robot controller based on a standard IRC5 robot controller [17]. The IRC5 has a modular design and contains two modules: (1) The drive module containing the electronics supplying the robot's motor with power and (2) the control module containing the control system electronics: main computer, I/O boards, and flash memory. These modules can be divided into separate cabinets. However, a standard IRC5 is delivered as a single cabinet with both a drive module and controller integrated. A single control system can control multiple robots, provided an additional drive module is added for each new robot added. Together the drive module and controller module provides the required electronics to drive and control the robot.

Figure 3.3: By removing the back panels from YuMi the integrated IRC5 can be exposed. Figure from [17].

The integrated controller of YuMi contains all electronics required to control the robot and comprises of a single drive module and control module. The integrated controller saves valuable working space and allows for a far more compact robot setup. Thus making YuMi quite portable and easy to install compared to a traditional robot cell with a far larger robot manipulator and external cabinets containing the controller. This portability is an important aspect of its intended collaborative use.

The embedded controller have two modes of operation: *Manual mode* and *Automatic mode*. In Manual mode, the operator controls the robot. The operator can manually jog the robot's arm, manually grip and jog the grippers and perform the operations provided in the FlexPendant. RAPID programs loaded to the controller can be stepped through but not

run normally. Tool Center Position (TCP) speed is limited in manual mode to 250 mm/s. In automatic mode, the operator is unable to control the robot via the FlexPendant. Automatic mode is intended for software-controlled motion and allows the software to control the robot in every supported function.

RobotWare-OS

RobotWare-OS, or just RobotWare, is the software running on the embedded robot controller. It is the robot's operating system and supports every aspect of the robot system [12]. In this subsection, RobotWare options and RobotWare Add-Ins will be presented, and the organization of user programs in RobotWare will be briefly reviewed.

RobotWare option
RobotWare options are optional functionalities that can be activated or deactivated on the robot controller. They provide extra functionalities for motion control, communication, systems engineering, or applications. Some specific RobotWare options require a license to be purchased from ABB, Externally Guided Motion, to be a reviewed later in the section, is such an example.

RobotWare Add-in
RobotWare Add-ins are self-contained software packages that extend the functionalities offered by the robot controller. While options are simply activated, add-ins need to be installed. RobotWare Add-ins can be developed by ABB or by a third party. RobotWare Add-Ins may require a license to be purchased, but in this project, no licensed Add-in was required.

Programming of ABB robots is done in the RAPID programming language, which is ABB's in-house developed high-level programming language for programming of industrial robots. RAPID programs are roughly split into two types of files: program modules and system modules. Program modules typically contain program-specific code and routines and have a .mod extension. System modules are used to define common, system-specific data and routines such as tools, and have a .sys extension. Program modules are organized into programs, and programs and system modules are organized into *tasks*. A task represents an application on the robot. Tasks can be connected to *mechanical unit group*. A mechanical unit group is a system parameter coupled to a mechanical device connected to the robot controller. It can be a separate robot arm, or it can be one of the arms on a dual-arm robot. YuMi, for instance, has two mechanical unit groups by default, "ROB_R" representing its right arm and "ROB_L" for its left. With the RobotWare option, Multitasking activated 20 tasks can run in parallel [12]. While multiple tasks can be run in parallel, only one task can be coupled to each mechanical unit group. The organization of modules into tasks is illustrated in fig. 3.4.

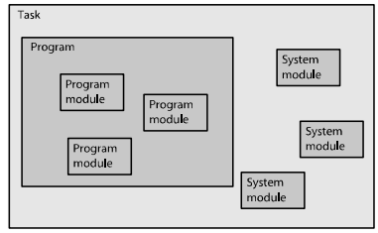


Figure 3.4: Program modules and system modules are organized in tasks. Figure from [12].

3.1.3 Auxiliary equipment

Smart Gripper

The Smart Gripper is a smart, multi-functional gripper from ABB specifically made for assembly and part handling. The gripper has one basic servo module, and two additional optional modules, vacuum, and vision [1]. The Smart Gripper have slots for vacuum on either side of the gripper, while the camera is mounted inside the palm. It is not possible to equip a gripper with two cameras. When equipped with a single vacuum, this vacuum can be placed arbitrarily on either side of the hand. Not accounting for vacuum placement, this leaves us with five possible configurations: vacuum + vision, vacuum + vacuum, vacuum only, vision only, and no additional modules. Grippers are swappable between arms. Machine drawings showing the dimensions can be found in appendix section 7.3.

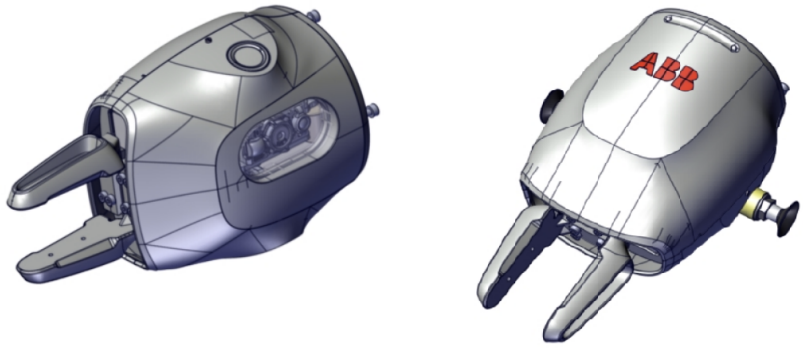


Figure 3.5: Figure shows pictures of CAD renderings showing the placement of the optional vacuums and camera. Render adapter from [18]

Servo module

All grippers come with servo a module. The servo module provides the functionality of

gripping. Finger movement and force can be supervised and controlled. The maximum gripping force is 20 N, maximum speed is 25 mm/s, and travel length is 25 mm per finger, 50mm total. More specifications supplied in table table 7.1 in appendix section 7.3. Load diagrams can also be found in appendix section 7.3.

Vacuum module
The vacuum module contains a vacuum generator, vacuum pressure sensor, and blow-off actuator. It provides the functions of suction and blow-off. Suction can be used to pick up objects, blow-off to place objects. The module can be monitored in real-time by the built-in vacuum sensor, which can be used to detect whether an object has been successfully picked up. The maximum payload is 150 gr.

Vision module
The vision module contains a Cognex AE3 In-Sight 2D camera. The module supports all functions of ABB Integrated Vision. The has a resolution of 1280x1024 pixels. More specifications supplied in table 7.2 in appendix section 7.3.

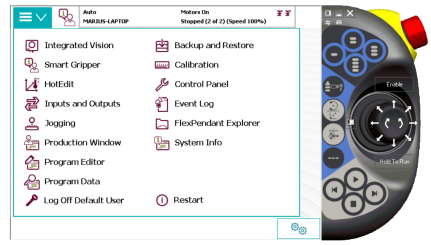
Communication
The Smart Gripper communicates with the robot controller over an Ethernet fieldbus. Grippers are assigned an IP address and are protected using IP30 protection. A RobotWare add-in ”SmartGripper” is available to facilitate the operation and programming of the gripper..

The YuMi in our robot system is equipped with two Smart grippers. The right gripper is equipped with the vision module, the left with a top-mounted vacuum module (top-mounted is pictured as the left side of the rightmost gripper in figure fig. 3.5. However, only the gripping functionality will be used.

FlexPendant

The FlexPendant is a hand-held device that can handle many of the functions involved with operating the robot system [19]. The FlexPendant is equipped with a touch-screen, traditional buttons, an emergency stop (pictured as a large red button in the images below), and a multi-directional joystick. The underlying OS of the FlexPendant is Windows CE, which is Micorsoft's operating System for embedded devices. On top of Windows CE runs graphical user interface (GUI), which is the environment the user interacts with. This GUI is mostly a multi-level menu system, pictured below in **??**. This menu system is navigated by the touch screen.

(a) The FlexPendant with the connector cable and some styluses.



(b) Screengrab from the virtual FlexPendant available in RobotStudio

Figure 3.6: The FlexPendant.

With the FlexPendant, the user can jog the robot, calibrate, adjust settings, change operation mode, start and stop programs and edit loaded RAPID programs [1]. The FlexPendant is particularly useful for starting and stopping program execution, debugging, monitoring of program execution, and jogging of the robot. Programmed prints or error messages will occur on the screen in addition to log messages informing the user of events in the system. By entering ”Production Window,” the user can monitor the execution of the running program. When in manual mode, the user can jog the robot both by jogging the joints directly in joint mode, or linearly from base coordinates in x-y-z directions defined by the robot base. When jogging in joint mode, the joystick can rotate three joints by vertical movements, horizontal movements, or by rotating the joystick head.

ABB RobotStudio

RobotStudio is an IDE developed by ABB for configuring and programming of ABB robots [20]. RobotStudio provides a virtual environment that simulates the physics of the robot. The software suit’s main propose is modeling, simulation, and offline programming of robot cells. RobotStudio supports both offline and online programming. When programming offline the user work on a simulation of the robot in the virtual environment and with a virtual controller. A virtual controller is an exact replica of the code running on the robot controller. When programming online the user program on the real robot and the real robot controller. RobotStudio can also simulate the FlexPendant when working offline in the virtual environment. RobotStudio requires Windows.

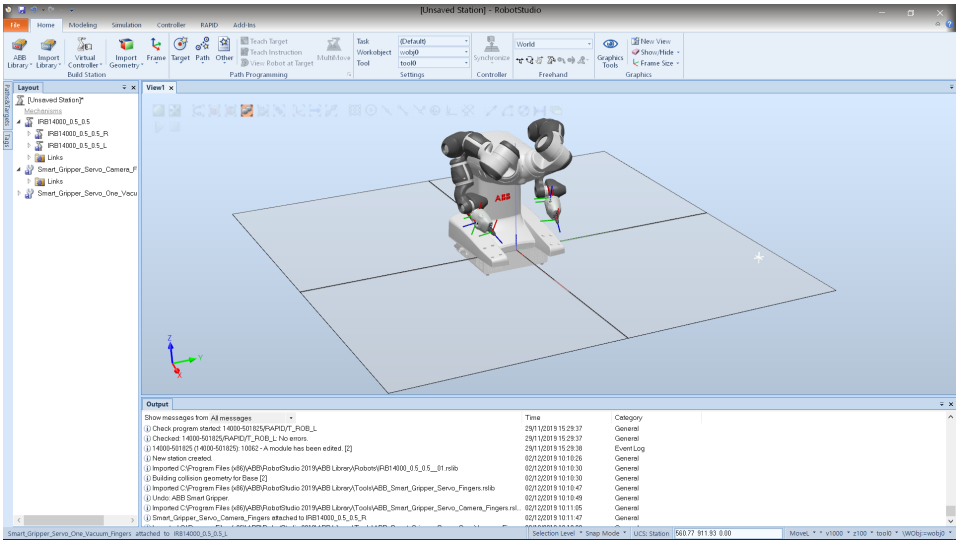


Figure 3.7: Figure shows a screenshot of the user interface of RobotStudio

3.1.4 External Control of YuMi

Lastly, two promising, available methods for external control of YuMi will be presented. We will review two types of interfaces for external communication with ABB robots: Robot Web Services (RWS) and Externally Guided Motion (EGM). fig. 3.8 below provides a useful illustration listing some functions provided by both interfaces. The illustration is from the 2018 ABB presentation of two C++ libraries for interacting with the interfaces, abb_libegm and abb_librws [21]. Support for some of the EGM functionalities for robots with more than six axes has since been included in RobotWare 6.09 and newer versions.

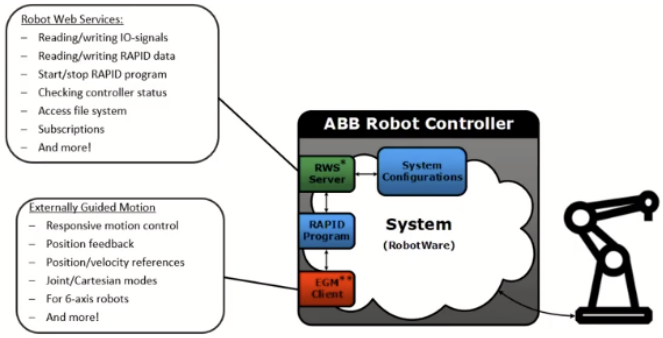


Figure 3.8: Figure showing the Externally Guided Motion (EGM) and Robot Web Services (RWS) interfaces. Figure from [21].

Both of RWS and EGM is used in the developed architecture. RWS is used for managing

and configurations purposes, and EGM for motion control. In this subsection, both EGM and RWS will be reviewed, starting with EGM.

Externally Guided Motion

Externally Guided Motion (EGM) is a licensed RobotWare option released with RobotWare 6.0. EGM enables external sensors or computers to control the motion of the ABB robot. EGM offers three features: EGM Position Stream, EGM Position Guidance, and EGM path Correction. When activating EGM, RAPID instructions for setup and control will become available to the user, and a set of system settings will be installed. This section will present the main functionalities of EGM, how these are invoked, and the data flow during EGM control. This section is based on the information found in the *IRC5 - Controller Software* user manual [12].

EGM Position Stream
EGM Position Stream provides functionality for the robot controller to stream the robot's current and planned positions of its mechanical units to external equipment. EGM position stream is available for UdpUC communication only. Its message contents are serialized and deserialized by the Google Protobuf definition. Streaming of positions can be done in up to 250 hz. Each motion task streaming data needs its own communication channel.

EGM Position Guidance
EGM Position Guidance provides functionality for using an external device to deliver position references for the robot controller. It provides a low-level interface to the robot controller, bypassing the path planner and communicating directly with the motor reference generator. This makes it suitable for when highly responsive robot movements are needed. Reading and writing positions to the motion system can be done every 4 ms (250 hz), and with a control lag of 10-20 ms depending on the robot type. This is the highest rate and lowest latency method for external motion control. EGM Position Guidance can be used both in joint mode and pose mode. Joint mode for joint position control, and pose mode for pose control.

- Pose mode is only supported for 6 dof robots.
- No interpolator functionality, meaning EGM Position Guidance do not perform linear movements.
- If a manipulator nears a singularity, the motion will be stopped. In this situation it will be needed to jog the robot out of the singularity.

EGM Path Correction
EGM Path Correction provides functionality for external robot mounted devices to generate path correction data. It delivers utilities for correcting a programmed robot path. It is at the moment only supported for 6 dof robots.

Features not available for use on YuMi will not be reviewed. Meaning no further information will be presented about EGM position guidance for pose control and EGM path correction. Interested readers are asked to consult [12] for information about these fea-

tures.

EGM states
The EGM control process is comprised of three states. This represents the state of the robot controller with respect to the execution of EGM commands and is useful for programming checks and fault handling. Below in table 3.2 the states are described. The tables are taken from [12, p. 331]

State	Description
EGM_STATE_DISCONNECTED	The EGM state of the specific process is undefined. No setup is active.
EGM_STATE_RUNNING	The specified EGM process is running. The EGM movement is active, i.e. the robot is moved
EGM_STATE_CONNECTED	The specified EGM process is not activated. Setup has been made, but no EGM movement is active.

Table 3.2: Different states of the EGM process [12, p. 331].

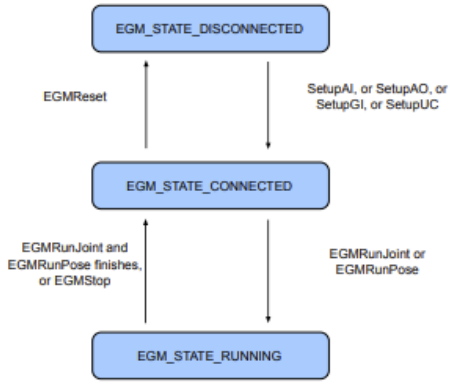


Figure 3.9: Figure illustrating the transitions between the EGM states [12, p. 331].

Above in fig. 3.9 the transitions between the EGM states are illustrated. A transition is triggered by the invocation of a specified type of function call. One of the Setup calls is required to move from EGM_STATE_DISCONNECT to EGM_STATE_CONNECTED, and opposite a EGMReset call is required to disconnect. The Setup command chosen defines the data source, and can be either outbound or inbound. This transition represents the connection or disconnection of an EGM device. The next transition between EGM_STATE_CONNECTED and EGM_STATE_RUNNING is triggered by calling either EGMRunJoint (joint mode control) or EGMRunPose (pose mode control), and the opposite transition is done by either finishing the requested motion or calling the EGMStop call.

EGM Position Guidance Control System
The EGM Position Guidance control system executes the motion commands received from

the external device. Requests and feedback are sent at a rate of 250 hz between the external device and the robot controller during position guidance. A new position reference is sent by UDP transport with the EGM Sensor Protocol (to be reviewed later) every 4 ms, feedback is returned at the same rate. The external device may be a Sensor or a UdpUc device and may send joint position or velocity commands. The underlying control system receiving these commands are showed in fig. 3.11.

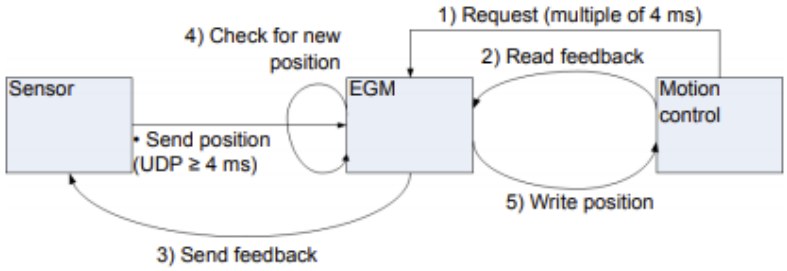


Figure 3.10: Data flow between the external device (depicted as a sensor), EGM and motion control. Figure from [12, p. 333].

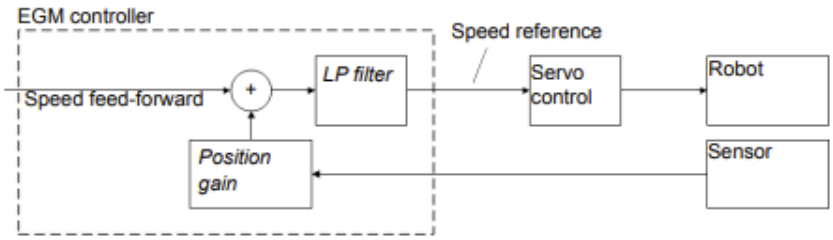


Figure 3.11: Block diagram of the EGM control system. External device depicted as a sensor. Figure from [12, p. 336]

The EGM controller executes the desired motion by issuing speed references to the servo motor control. The control loop is based on the following relation between speed and position. [12, p. 343]:

$$\text{speed} = k * (\text{pos\_ref} - \text{pos}) + \text{speed\_ref} \tag{3.1}$$

$speed$ denotes the desired speed sent to the servo motor controller, $speed_ref$ denotes the reference for this speed. Likewise, pos_ref denotes the reference position, and pos the desired position. The references represent the actual position or velocity of the joint in question. The K-factor k is composed of two systems constant that are modifiable by the user.

$$\text{k} = \text{PCG} * \text{PPG} \tag{3.2}$$

PPG denotes the Proportional Position Gain is a system configuration setting. It is the gain desired for proportional gain control (P control) of the position. PCG denotes the Proportional Correction Gain and is specified in RAPID modules for run time instruction calls. PCG is the gain associated with pose correction.

The issued speed references can be subjected to corrections by the EGM application. Corrections are divided into three system levels, listed in table 3.3. Raw corrections are direct corrections to the tp-be issued speed reference. Desired joint positions sent from the external device are a type of level 0 correction. Level 1 correction is extra filtering, which can be activated when setting up the EGM connection. Level 1 correction is known to introduce extra latency and delays [12, p. 343]. Path Correction is a Level 2 correction.

Level	Description
Level 0	Raw corrections made just before the servo motor control.
Level 1	Extra filtering on the correction.
Level 2	Path Correction.

Table 3.3: The three system levels in which corrections are applied. Adapted from [12, p. 343].

EGM Sensor Protocol
The messages sent between the external device and the robot controller are sent according to the EGM Sensor Protocol [12, p. 339]. The EGM sensor protocol is designed for high-speed communication with minimal overhead. The protocol uses UDP for transport, and Google Protocol Buffers (protobuf) for serialization and de-serialization of the data. The external device is playing the role of the server, waiting for a first message from the robot controller before transmitting any messages. The robot controller, acting as the client, sends this first message when EGM Position Guidance is called by the robot controller via a RAPID instruction. Messages can be sent in both directions independently of each other after the first message. The protocol features no built-in synchronization of requests and responses or handling of lost messages.

Initiating Motion With EGM
Lastly, An EGM motion has to be initiated from a *fine point*, which is a zonedata specifying how a position is determined reached. The implication of this is that an EGM motion cannot be performed during a non-EGM motion is being performed, and requires the manipulator in question to be standing still to start. Below in fig. 3.12, a flowchart showing how an EGM motion process can be initiated.

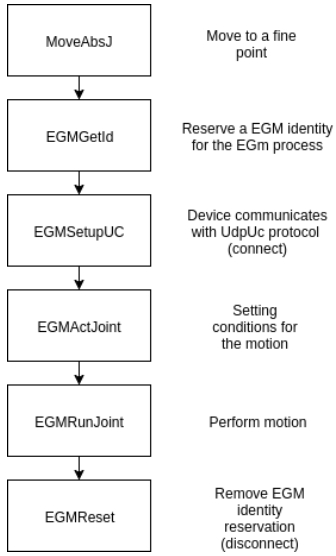


Figure 3.12: Flow chart of an EGM motion process.

Robot Web Services

Robot Web Services is a networked, HTTP wrapped API for communication with the IRC5 controller [22]. Web services are platform-agnostic, meaning any connected device can communicate with the robot controller. The web services are designed after the REST architecture style, meaning URLs identify resources instead of web locations. Responses from the controller can be delivered in JSON or XML format, with XML as default. The web services provide means to manipulate the robot in most aspects that normally would require physical access to the robot, from an external device. Ability to turn motors on/off, fetch system info, fetch variable values, fetch or modify settings, command motion are some examples of the services provided. It is also possible to inspect information about the robot through a web browser by typing the address of the robot in the address field. ”¡robot ip¿/rw?debug=1” will for instance provide debug information. While the HTTP is typically used as the application protocol, WebSockets are also supported. Data transport is done by the TCP transport protocol.

3.2 ROS2 Integration of YuMi

Having presented the robot to be controlled, the focus will now move to the integration of YuMi with ROS2. Before presenting the developed architecture, the structure of the presentation will be established and a system overview provided.

3.2.1 System Overview

The architecture is composed of two subsystems: a robot side and a ROS2 side. The robot side resides inside the embedded robot controller, running on the RobotWare OS. The ROS2 side resides in an external computer running Ubuntu 18.04 and ROS2 Dashing. The computer is connected to the robot controller through an Ethernet connection and communicating with the robot controller both via RWS and EGM. An overview of the system is shown in fig. 3.13.

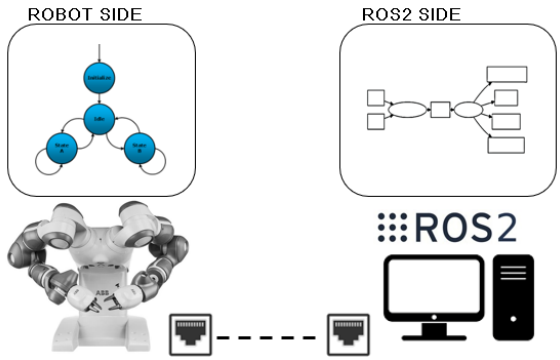


Figure 3.13: System overview

3.2.2 Structure

With the software being distributed over two physical locations, the review of the architecture developed will likewise be structured in two parts: the robot side architecture will be presented in section section 3.3 and the ROS2 side in section **??**. The architecture will be presented with the following structure: firstly, the robot side configurations and applications will be presented. Further, moving to the ROS2 side, the libraries used to communicate with the robot controller will then be presented, followed by a presentation of the ROS2 framework and packages used to form the architecture of the ROS2 side. Continuing, the robot-specific implementations of these will be presented alongside the main components of the ROS2 architecture. With all components presented, their interactions will be reviewed, and a small demo of the API will be presented.

3.3 Architecture - Robot Side Subsystem

This section will present the architecture of the developed robot side subsystem of the ROS2 integration. Underlying components and modules will be reviewed, and the execution flow will be described. Lastly, before turning to the ROS2 side subsystem, an externally controlled minimal move and grab operation will be presented to illustrate how the robot can be externally controlled via the robot side subsystem.

3.3.1 Robot side - The Link Between ROS2 and the Hardware

ABB's robot controllers are a closed system and therefore do not provide access to the lower layers of the controller software. While API's for interacting with the upper layers of the software are provided, it is not possible to interact with the hardware and motor control systems. The robot side part of the system should hence act as a link between the exposed API and the ROS2 application on the desktop side.

StateMachine Add-In

The robot side RAPID programs are loaded from the StateMachine Add-In developed by ABB [23]. This Add-In provides a state machine implementation in RAPID code. When installed, the Add-In installs the RAPID modules and system files, connects modules to tasks and mechanical unit groups, adds its I/O signals to the controller, and loads configurations to the system. After the installation, most of the necessary system modifications are performed.

The state machine provides means for communication with an external component. The external component can be any program in any language that communicates via the supported channels and resides in a connected external device. In our system, this external device is a ROS2 node. The state machine is intended to be used together with Robot Web Services (RWS) and, optionally, Externally Guided Motion (EGM) [24]. Naturally, for EGM communication, the Externally Guided Motion RobotWare option is required.

States of the State Machine

The state machine is a finite state machine, meaning it got a finite number of possible states with a finite number of defined transitions between them. The state machine is controlled from the external device by listening to changes on some installed IO signals. The IO signals are typically coupled to requesting transition to different states. The state machine features five states, shown in table 3.2 below. The states represent the four modes of operation and an additional unknown state for fault handling. The modules comprising the state machine are listed in table 3.5 and the execution flow of the state machine is visualized in fig. 3.15.

State	Description
Idle	Idle state
Initialize	Initialization state
Run RAPID routine	Running RAPID routine state
Run EGM motion	Running EGM routine state

Table 3.4: States of the state machine [25]

Module	Description
TRobMain	Contains the main, initializes other modules and runs the state machine.
TRobEGM	Contains all functions and variables related to EGM mode.
TRobRAPID	Contains all functions and variables related to RAPID mode.
TRobSG	Contains all functions and variables related to Smart Gripper operation.
TRobUtility	Contains commonly used functionalities and helper functions.
TRobWatchdog	Watchdog supervising the system.

Table 3.5: Program modules of the state machine [25]. TRobMain contains the ”main”.

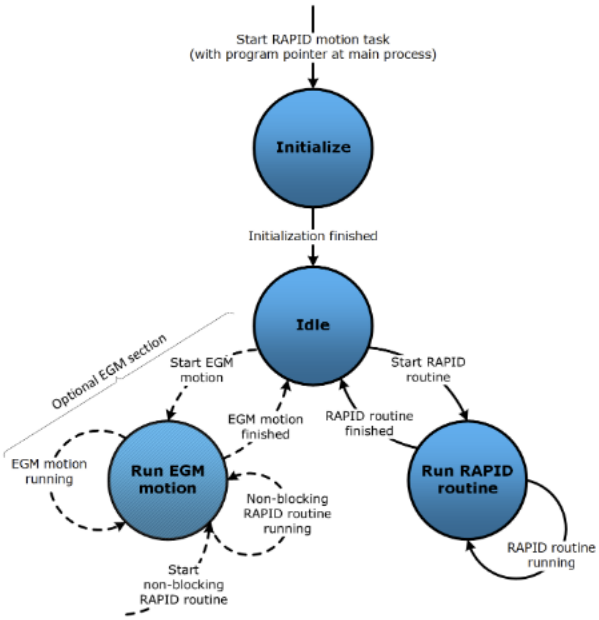


Figure 3.14: Execution flow of the state machine. Figure from [25].

Module Organization

The program modules are loaded into one task, which is then coupled to one mechanical unit group. Under installation, it is therefore loaded one set of modules for each mechanical unit group, meaning deployment of one state machine for each mechanical unit group. In the case of YuMi this means two state machines, one for each arm. This requires the robot controller to run the two state machines in parallel, and hence the RobotWare Add-In Multitasking is required.

Execution Flow

The four operational states represent four different modes of operation possible with the state machine. Upon startup, the module ”TRobMain” is executed, and the state machine enters the *Initialize* state where it initializes the other modules. There exists no condition

for it to reach the state, and it is the first state to be reached, and when left, the state machine can never return to this state.

The *Idle* state is the base mode to which the state machine will always return after successfully completing an operation in another state. Upon start, after initialization, the state machine transitions from the *Initialize* state to the *Idle* state, and the ”TRobMain” module executes an infinite loop. The loop contains a test checking the state variable containing the state of the state machine and compares this to two cases: ”STATE_RUN_RAPID_ROUTINE” and ”STATE_RUN_EGM_ROUTINE”. Both cases contain a function call to run the respective routine located in respectively ”TRobEGM” and ”TRobRAPID”. The loop is listed in XX. Which state is entered depends on the IO signal received from the external device. *Idle* is the only state reachable from all states.

```
WHILE TRUE DO
  TEST current_state
    CASE STATE_RUN_RAPID_ROUTINE:
      runRAPIDRoutine;

    CASE STATE_RUN_EGM_ROUTINE:
      runEGMRoutine;

    DEFAULT:
      ! Do nothing.
  ENDTEST

  WaitTime 0.01;
ENDWHILE
```

Listing 3.1: The infinite loop run in the Idle state.

In the state *Run RAPID routine*, the state machine can run predefined RAPID routines loaded to the robot controller. The RAPID routine can be transferred to the robot controller during runtime. When finished the state machine return to state *Idle*. In *Run EGM motion* the state machine connects to the predefined UdpUc device containing the EGM server. If a successful connection is made, it executes whatever joint references streamed by the external EGM server. If no commands are received between the connection is made and a defined time limit, the state machine will stop and output an error. If commands are received but no motion is required, it will, after the same defined time waited, stop listening for commands and return to *Idle*.

Additional functionalities not connected to a particular state is defined in ”TRobSG”, ”TRobUtility” and ”TRobWatchdog”. ”TRobSG” provides routines for operations with the Smart Grippers, containing functionalities for initialization and calibration, jogging, gripping, and suction control. ”TRobUtilities” contains helper functions used by the other modules, and ”TrobWatchdog” handles stop requests and some error handling.

Interrupt Driven Actions and Transitions
All transitions from *Idle* are handled by interrupts. Interrupts are handled via traps listening on different IO signals. If an interrupt is to be caused for both a LOW (zero) or a

HIGH (one), two traps are provided for that IO signal. Upon receiving an interrupt triggered IOsignal change coupled to a change in the mode of operations, i.e., moving to either *Run EGM motion* or *Run RAPID routine*, the first thing happening is that the appropriate trap is called. This trap changes the state variable to the appropriate state, which will then be discovered and acted upon in the infinite loop in TRobMain. Transitions from *Run EGM motion* and *Run RAPID routine* back to *Idle* are not necessarily handled by interrupts. EGM mode can be exited by an interrupt coupled to the receiving of a stop signal. This is, however, not possible for RAPID mode.

Smart Gripper operations are also interrupt-driven. Smart Gripper operations are non-blocking and cause no state change, meaning they can be called with minimal disruption to the execution flow. Smart Gripper operations can be executed from any non-blocking phase of the run time. The operations have been confirmed to be callable during EGM motion, which is in agreement with the non-blocking nature of the execution of the motion commands. Smart Gripper operations have, however, not been verified to be callable during the running of RAPID routine in the *Run RAPID routine* state.

Execution Flow of A Minimal Move and Grab Operation
To illustrate how the state machine can be used, a minimal move and grab operation will be presented. The operation features the use of RWS for starting of the state and checking if the robot is ready to be controlled, RWS for the operation of the grippers, and EGM for control of the robot arm to be used. The operation is shown as a flow diagram below in fig. 3.15.

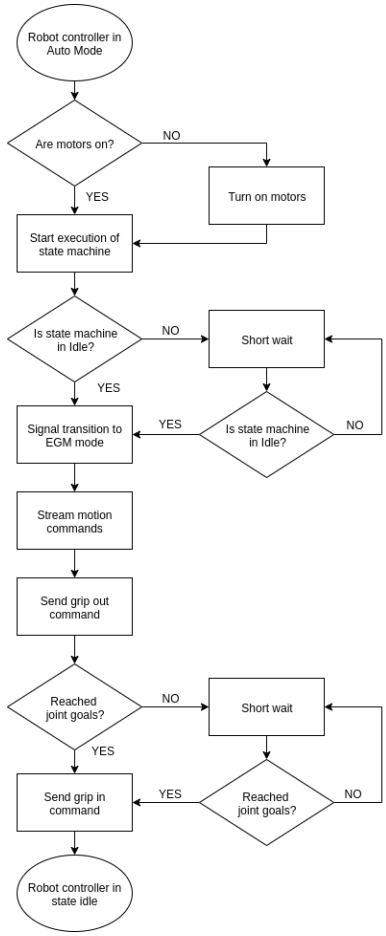


Figure 3.15: Flow chart showing how an external device can use the state machine to execute a minimal move and grab operation.

For the state machine to be able to move according to received motion references, the robot controller needs to be in Auto mode, the motor must be switched on, the state machine must be started and have reached state *Idle*. RWS cannot be used to change the operation mode from manual to Auto, so this must be configured manually by the FlexPendant. When in Auto mode, commands to turn the motors on and start execution of the state machines can be issued. The external device can choose to poll the state machines regularly until they reach *Idle*. When in *Idle* An EGM connection can be made, and commands can be streamed. This will cause a change in state, putting the state machines into EGM mode, performing streamed commands. While moving the arm, the grippers can be opened, and upon reaching the desired location, closed, and thus finishing the simple move and grab operation.

3.4 Architecture - ROS2 Side Subsystem

This section presents the architecture of the developed ROS2 side subsystem of the ROS2 integration. The section will begin by reviewing how ROS2 will interface against the controller, then the building blocks of the ROS2 subsystem will be presented, followed by a presentation of the architecture. Lastly, before turning to the evaluation of the complete systems performance, a minimal move, and grab ROS2 application is presented.

3.4.1 Interfacing Against the Robot Side Subsystem

For the ROS2 side to be able to control the robot using the EGM control system, it must be able to command the state machine to change state to EGM mode. The transition occurs when the sate machine registered the required change in the coupled IO signal. Hence, the ROS2 side must be able to deliver this IO signal through a supported means of communication with the robot controller. Robot web services are chosen for this purpose. Additionally, the ROS2 side must be able to communicate using the EGM sensor protocol to successfully establish a connection with the robot side. As the robot controller is configured to transmit its state as the first message, the ROS2 side must include an EGM server that will establish a connection with the robot controllers EGM client when the state machine has transitioned to EGM mode. As all previously mentioned, there exist two open-source C++ libraries made by ABB in 2018, which supplies the user with abstractions simplifying the use of the RWS and EGM interfaces. While the libraries are hosted as ROS packages, the libraries themselves contained only C++ code, and only a small effort was needed to convert these to ROS2 packages. Seeing as these libraries will provide the needed functionalities for communication with the state machine, the architecture presentation will start with a brief presentation of the library's organization, classes for abstraction, and functionalities.

abb_librws

abb_librws [25] is a C++ library for interfacing with ABB robot controllers supporting Robot Web Services. It is layered in a typical object-oriented style, as shown in fig. 3.16. A presentation of the library's classes is provided in fig. 3.16.

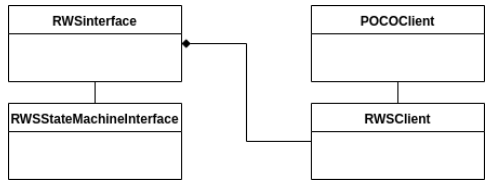


Figure 3.16: Classes of abb_librws.

The lowest layer of the library is the *POCOClient*. Using the POCO C++ library, this class sets up and manages the HTTP and WebSocket. *RWSClient* inherits from *POCOClient* and

implements the RWS protocol. Provides methods for interactions with RWS services and resources. *RWSInterface* encapsulates and *RWSClient* instance and wraps the methods in a more user-friendly API. *RWSStateMachineInterface* inherits from *RWSInterface*. It is a special case of a *RWSInterface* for interactions with the StateMachine Add-In. It is aware of the state machine specific RAPID variables, routines and system configurations, and further wraps *RWSInterface* methods to make the state machine specific.

abb_libegm

abb_libegm[26] provides resources for setting up an EGM server to communicate with an EGM client located in the robot controller. It provides an EGM interface enabling streaming of motion references to the robot via EGM Position Guidance and to receive feedback of robot state via EGM Position Stream. Both joint mode and pose mode control are supported.

It is layered in a typical object-oriented style, as shown in fig. 3.17.

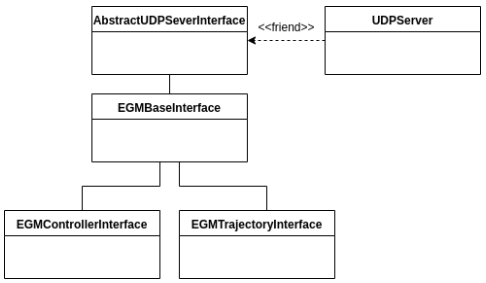


Figure 3.17: Classes of abb_libegm.

UDPServer supplies the low-level abstractions. It handles the UDP server and manages the communication loop. *AbstractUDPServerInterface* is an abstract interface for used by the derived classes to interact with the *UDPServer*. *EGMBaseInterface* implements a base EGM interface. It inherits from *AbstractUDPServerInterface*. *EGMControllerInterface* is derived from *EGMBaseInterface*. This is the abstraction applications using the library interact with. It implements an EGM interface for the writing of commands and reading of state, providing methods to be used by an application in an external control loop. *EGMTrajectoryInterface*likewise but for writing of command queues forming trajectories.

3.4.2 ros_controls

The developed control architecture is developed on top of the ros2_control framework for robot control. ros2_control is a stack of packages residing in ros_controls [27]. ros_controls is a collection of ROS stacks that provides tools, hardware abstraction, and controllers for developing end-to-end real-time control systems for robots. The collection is written in C++ and provides robot-agnostic base classes providing a standardized interface against

the hardware and each other. From these, robot-specific implementations can be derived. ros_controls deliver the packages and tools to fill the ”black box” between the application and the robot hardware, illustrated below in fig. 3.18. fig. 3.19 contains a more detailed view of the contents of the ”black box”. ros_controls contains mostly tools and stacks for ROS1, however, some packages and tools are available for ROS2. Moving on, only the ROS2 packages of ros_controls will be considered. Illustrations from the ROS1 documentation have been used and will be used throughout. Deviations between the illustration and the ROS2 version will be notified.

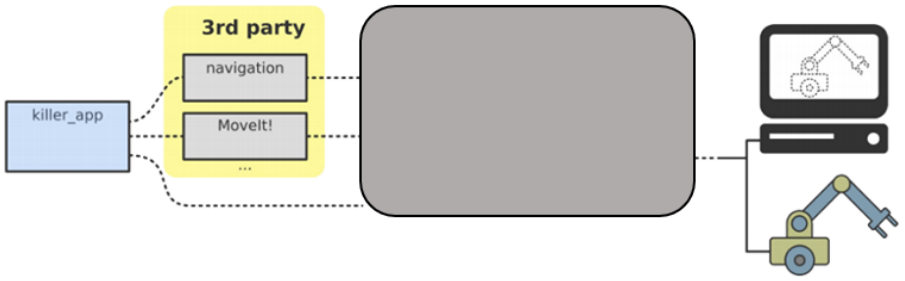


Figure 3.18: illustration showing the ”black box” which ros_controls utilities are intending to fill.

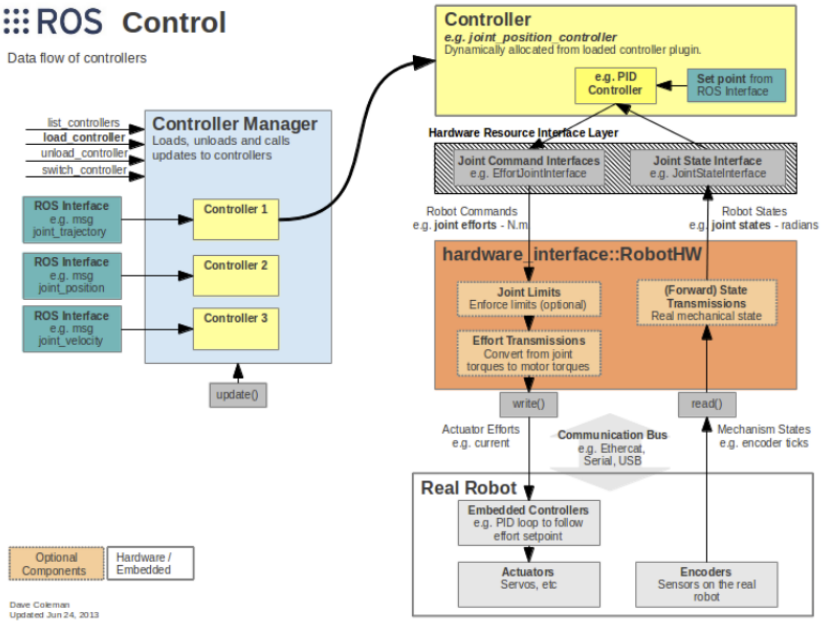
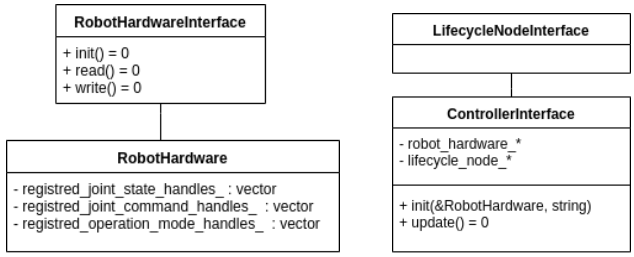


Figure 3.19: Diagram showing how to different components from ros_controls can be used together to form a end-to-end solution. Figure from [28].

Pictured above in fig. 3.19 is a diagram showing how the ros_controls packages can be set up to form an end-to-end control solution. ros_controls can be thought of as a framework for developing hardware abstractions and control systems for robots in ROS, containing stacks of packages responsible for the different aspects or phases of the control cycle.

ros2_control
ros2_controls is the stack of packages forming the backbone of ros_controls. ros2_control's main contribution is the controller_manager and base classes from which robot or controller specific instances can be derived. The classes are shown in fig. 3.20.



(a) RobotHardware **(b)** ControllerInterface

Figure 3.20: ros2_control's *RobotHardware* and *ControllerInterface*

The hardware interface is used to form a hardware abstraction layer between ROS and the robot hardware. A base class from which robot specific implementations can be derived is provided in *RobotHardware*. It is provided in the hardware_interface package and is depicted in fig. 3.19 as *RobotHW*. The hardware interface is responsible for abstracting away the hardware. It contains hardware-specific methods for communicating with the hardware, which is then wrapped in a simple read() and write() API to be used by the framework. As a typed interface, it is easier to maintain, provides easier introspection, and allows controllers to be hardware-agnostic.

At this moment, there exists no hardware resource interface layer in the format depicted in fig. 3.19. The controller instead is loaded with reference to the underlying *RobotHardware* from the robot specific implementation. Both states and commands are registered in this underlying class, meaning the controller can access the states and write commands directly to the memory locations from which the class transmits its commands to the robot. By having the robot specific hardware abstraction as a *RobotHardware* typed class, the robot-agnostic properties are upheld with minimal overhead on the passing of data, allowing for low latency control cycles.

The controller manager is responsible for handling the lifecycle of the controllers. It provides methods for loading, configuring, activating, and use of controllers. The controller manager provides a fast and efficient solution to using several controllers in a single control cycle. Typically at least two controllers are loaded: a controller of choice for issuing of commands, and one for publishing of the received robot state to a state topic. F.ex. a

joint position controller coupled with an joint state controller.

The controllers are loaded and managed by the controller manager but have direct access to the memory locations of the read robot states and the commands to be written. The controllers update the command to be sent based on the fed back robot states and the update law implemented. ROS2 applications looking to control the robot interact with the robot controller via one of the available interfaces, typically action based or topic-based. A typical controller needs to be fed desired values for the entities it controls via one of the interfaces. The interfaces are depicted in fig. 3.19. As controllers are robot-agnostic, controllers can be made for one robot but used on other robots in accordance with the ROS philosophy of reusable code. Similar to the hardware abstraction, controllers must inherit from a supplied base class. The ros_controls stack ros2_controllers contains only two pre-made controllers, a *joint_trajectory_controller* and a *joint_state_controller*. All controllers must inherit from *controller_interface* for the controller_manager to be able to manage the controller, meaning all custom controllers must be derived from this class. The *controller_interface* is found in ros2_control.

3.4.3 Main Components

Having reviewed the three main building blocks from which the ROS2 side subsystem will be built upon, abb_libegm, abb_librws and ros2_control, the ROS2 side subsystem will be presented.

The classes comprising the developed core components are listed below in table 3.6. A GitHub repo hosting all packages used is found at [29].

Class	Description
AbbEgmHardware	Handles joint control via EGM
SgControl	Handles control of the SmartGrippers via RWS using
YumiRobotManager	Manages the YuMi via RWS using
JointPositionController	Controller for control of joint position

Table 3.6: Classes comprising the core components of the architecture.

AbbEgmHardware

AbbEgmHardware is the robot's robot-specific hardware abstraction layer derived from the ros2_control's *RobotHardware*. YuMi features two arms, which will be provided an *AbbEgmHardware* hardware abstraction layer each. The class is created to be configured by loading arm parameters from a parameter server, meaning the class can represent both arms. *AbbEgmHardware* is intended to provide a code-representation of the physical arm, meaning two instances must be run. This involves two EGM servers to be run simultaneously, each communicating with a unique client residing on the two running state machines. A connection is made between the arms state machine EGM client and the

AbbEGmHardware instances EGM server, meaning a separate connection is required for each arm, as shown in fig. 3.21.

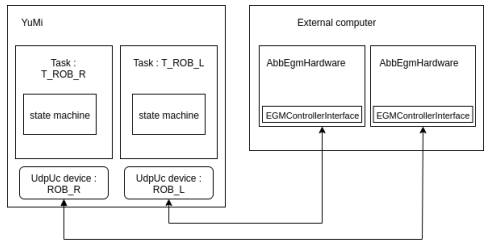


Figure 3.21: A unique AbbEgmHardware instance, state machine and egm connection is required for each arm.

Below in fig. 3.22 the class is visualized in an UML class diagram with a selection of its most important methods and attributes. Being a *RobotHardware* typed hardware abstraction layer, the class is required to feature at least three methods: init(), read() and write(). Its init() initialization method handles the setup. The init() method loads the needed information about the robot arm, creates and registers the handles for, among other attributes, joint states and joint commands, and enters a infinite loop terminated when the EGMControllerInterface is initialized. EGMControllerInterface is determined initialized when its underlying EGMBaseInterface inherited function isInitialized() return true if. It returns true only if the underlying server was successfully set up and is receiving messages from the robot side EGM client. The init() is blocking until a successful connection between the robot side EGM client and the EGM server has been made.

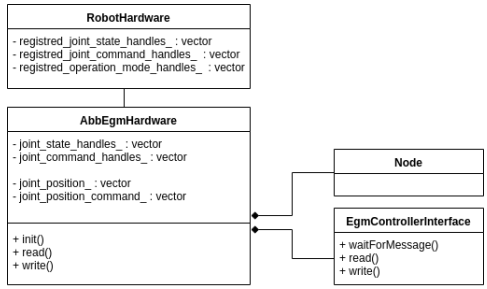


Figure 3.22: UML class diagram of AbbEgmHardware.

The read() method of *AbbEgmHardware* waits until a message from the EGM client is received or a timeout limit is reached. If a message is received within the timeout limit, the state is read using the *EGMControllerInterface* read() method into a input-container from abb_libegm and then copied to, among other attributes, to the joint_position vector. The read() method of *AbbEgmHardware* is therefore potentially a blocking call for a predefined time limit. Meaning, if included into a control loop, the control loop will be throttled to 250 hz. As will be seen later, this will the case for for YuMi. The write() method copies

the contents of registered joint_position_command to a output-container which is sent using *EGMControllerInterface* write() method. When data is copied from the input/to the output container it is deserialized/serialized in accordance with the EGM sensor protocol.

The abb_egm_hardware is run by a dedicated executable. The executable runs an internal control loop spinning at 250 hz, where each iteration starts with a message being read, a command being calculated and command being written. No spin rate limitations are put on the control loop, as it is throttled to the communication frequency of the EGM connection by the blocking in the read() method of *AbbEgmHardware*. A simplified version is shown in listing 3.2 below

```
int main()
{
  **Necessary other setup.**
  robot->init();

  **controller_manager instance created.**
  **Controllers are loaded, configured then activated.**

  // The control loop.
  while (rclcpp::ok())
  {
    robot->read();
    controller_manager.update();
    robot->write();
  }

 **Teardown.**
  return 0;
```

Listing 3.2: Executable running running the instance of of AbbEgmHardware, the controller manager and the arms control loop.

JointPositionController

The *JointPositionController* is an joint position controller which inherits from the controller_interface base class, enabling it to be managed by the controller manager. It contains all the necessary lifecycle methods, including methods for error handling and teardown. The class is visualized in the UML class diagram shown in fig. 3.23. For simplicity, most of its lifecycle methods is omitted.

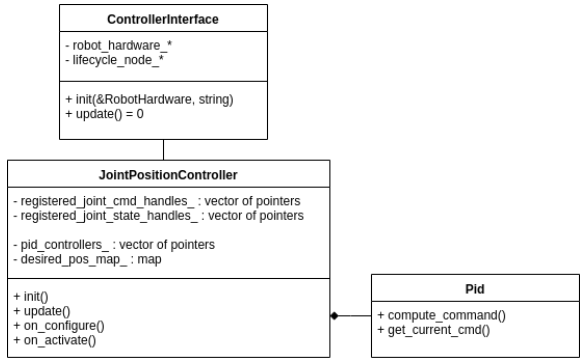


Figure 3.23: UML class diagram of JointPositionController.

The controller is configured by loading its needed parameters from the parameter server, and is completely robot-agnostic. Its update() function applies the control strategy. A simple control scheme with a PID-controller is applied:

```
For each joint:
  if(actual position != desired position)
    Find deviation between desired position and actual position.
    Denote this value e.
    addition = pid->compute_command(e)
    commanded position = actual position + addition
```

Listing 3.3: The joint position control scheme.

The controller calculates the computed command using a PID controller. The PID gains are loaded from the parameter server during configuration. While possible, and intended to be able to apply PID-control in the joint_position_controller, in the current system it is configured to operate functionally as a P-controller by loading integral and derivation gains set to zero.

The joint_position_controller have a topic based interface towards the rest of the ROS2 system. It listens to the topic /JointCommands for desired joint positions, executing a callback updating the desired_position_map_ with the received desired positions for each message received.

YumiRobotManager

The state machine requires an IO signal to enter its EGM mode. This presents the need for a component responsible for ensuring the robot controller and the state machines are ready to be operated in EGM mode. Besides writing of IO-signals this puts requirements for other functionalities like reading and modifying controller status, reading of IO signals and the ability to start and stop the execution of the state machines. The class utilises methods from abb_librws to handle such operations. The class is visualized in the UML class diagram shown in fig. 3.24

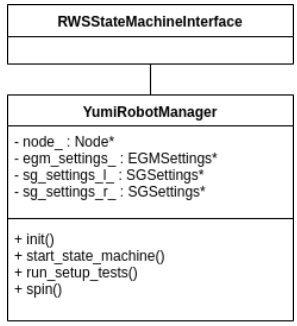


Figure 3.24: UML class diagram of YumiRobotManager.

The RWS operations offered are split into two categories: a automatically performed upon launch, and a set offered to other nodes as ROS2 services. Below in fig. 3.25 a overview over the two categories is shown.

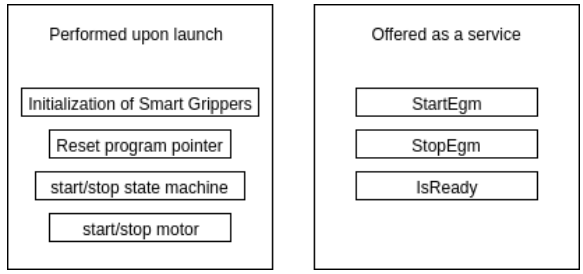


Figure 3.25: YumiRobotManager provides Robot Web Services operations both as methods and ROS2 services..

YumiRobotManger is launched via a dedicated executable. The executable runs its initialization method init(), signals the state machine to start with the start_state_machine() method and runs setup tests with the method run_run_setup_tests(). After the initialization motors are ensured to be on and the program pointers ensured to be reset. The start_state_machine method starts the state machines and ensures the state machine is in state *Idle* before it returns. The setup tests is at the moment limited to calibrating the Smart Grippers and moving them to the closed position. The state machine operations called via RWS is shown in fig. 3.25.

YumiRobotManager offers three state machine operations as ROS2 services: starting EGM mode, stopping EGM mode and polling of state machine state.

SgControl

The final class to be reviewed is *SgControl*. *SgControl* offers a code-representation of a Smart Gripper. It is set up as a node running a action server, offering a Grip action on

a Smart Gripper through a *RWSStateMachineInterface*. The Grip action is defined in a manner to handle both opening (gripping outwards) and closing (gripping inwards) of the gripper. *SgControl* contains no polling method for determining the state of the gripper. The feedback provided for the action is a countdown from 1.5 seconds, reporting ”progress” in 10% increments. 1.5 seconds was determined from simple time measurements indicating both grip out and grip in operations to have a duration between 1.2 and 1.5 seconds. A UML class diagram is shown in fig. 3.26.

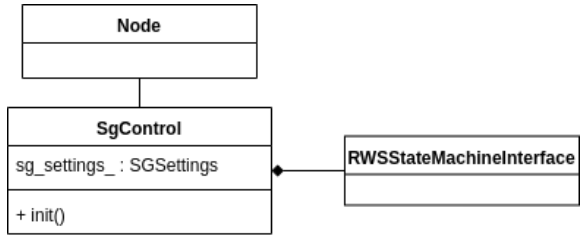


Figure 3.26: UML class diagram of SgControl.

SgControl is launched from dedicated executable. The executable initialize the class and spin the action server. *SgControl* assumes the state machine to be running.

Computational Graph

Below in fig. 3.27 the computational graph of the architecture is shown. Every presented class except the *YumiRobotManager* is launched in two instances, organized in two namespaces, namespace ”/r” for the right arm and ”/l” for the left arm. *YuMiRobotManager* managing the robot controller, whom there exists only of on the robot, is in the global namespace and in a single instance. This is in line with the previously hinted at design philosophy.

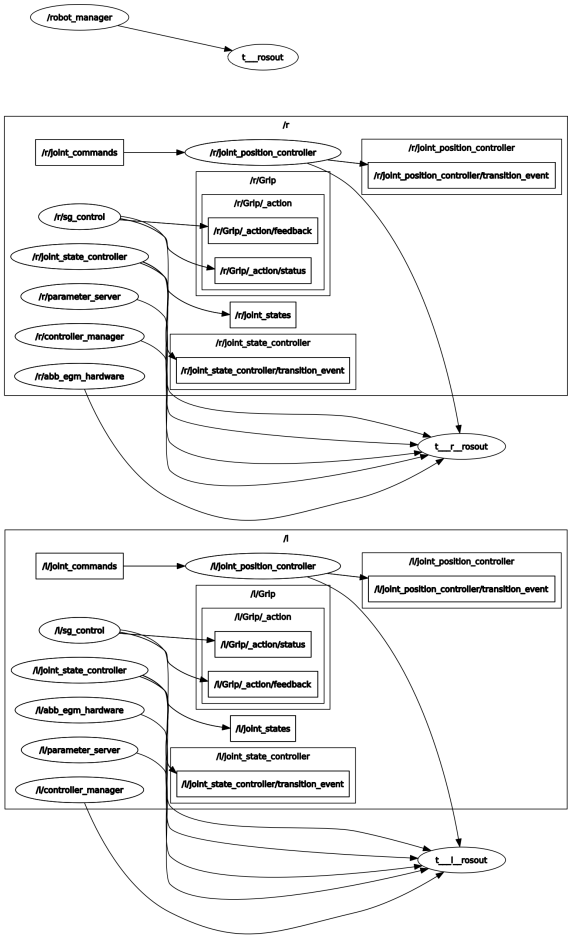


Figure 3.27: The computational graph of the architecture.

In addition to the presented classes, the computational graph features a joint_state_controller and parameter_server fro each arm. The joint_state_controller used is an implementation available in ros2_controllers, found in [30]. The parameter_server is loaded with arm and controller specific parameters at deploy time and offers this data as services to rest of the namespace. It is used to allow *AbbEgmHardware* to represent both arms, and to enable the joint_position_controller to be robot-agnostic.

The PID controller, *joint_position_controller* and the parameter server is adapted from freely available implementations found on GitHub.

A Minimal Dual-Arm Move-and-Grab ROS2 Application

To demonstrate the API of the architecture, a minimal ROS2 dual-arm move-and-grab application will be presented. The application illustrate how a ROS2 application can control YuMi with the developed architecture via its service, action and topic-based API's. In the application, instances of client classes for use of *SgControl* actions and *YumiRobotManger* services are used. The clients are concerned with the use of the architecture, and by not being a inherent part of the control architecture these were excluded from the presentation. Interested readers can find their implementation on the architectures repository on GitHub [29] in package "rws_clients".

Pseudocode hints:

- The callback of the nodes subscriptions updates a global variable containing the current position of the arms joints.
- move_point(<goal>, <namespace>) is call to a simple publish function. The function simply generates a message with the new goal which is published to <namespace> /join_commands

```
int main(){
  ** A node for each namespace is created.
  Each node sets up a subscription to the its namespaced
  joint_states topic and sets up to publish on the
  namespaced joint_commands **

  ** Clients for use of SgControl actions and
  YumiRobotManager services constructed
  and initialized **

  while(!yumi_robot_manager_client->robot_is_ready()) sleep(1);
  yumi_robot_manager_client->start_egm());

  /* open both grippers */
  grip_action_client_r->perform_grip(0)
  grip_action_client_l->perform_grip(0)

  goal_r = {42.1, -118.4, -92.1, 46.5, -143.6, 42.5, 36.1};
  move_point(goal_r, "/r");
  **waits until goal is reached**

  goal_l = {-39, -104.1, 80, 8, 31, 93, 57};
  move_point(goal_l, "/l");
  **waits until goal is reached**

  /* close both grippers */
  grip_action_client_r->perform_grip(100)
  grip_action_client_l->perform_grip(100)

  yumi_robot_manager_client->stop_egm());
  printf("Task completed"\n);
  return 0;
}
```

Listing 3.4: Minimal assembly ros2 application controlling YuMi through the developed architecture.

Chapter 4

Evaluating the Performance of the Developed Architecture

So far, the preliminaries for the architecture, the robot system on which it is installed, and the two subsystems of which the architecture is composed have been reviewed. The previous sections have been scoped to present the necessary details to enable an evaluation of the developed systems' performance. This section will present how good and bad performance can be defined, define key performance indicators, how these will be measured, and finally, the experimental setup in which the tests and measurements are conducted.

4.1 What Will be Evaluated

A distributed system for the external ROS2 based real-time control of the IRB14000 ”YuMi” has been developed. The system is distributed with a robot side component connecting the ROS2 components to the EGM control system. The ROS2 side contains a hardware abstraction enabling the EGM communication, RWS components for managing the robot and its grippers, and a proportional gain controller for joint position control of its arms. The complete architecture provides two control systems: a closed-loop system for the joint position control of YuMis joints, and an open-loop system for the gripping functions of its Smart Grippers. In the evaluation of the system, only the closed-loop joint position control system will be evaluated. Moving forwards, *system* or *the developed architecture* refers to the closed-loop joint position control system

One of the goals for the joint position control system was to achieve a somewhat real-time response from the system. A response is denoted as a commanded joint movement being actuated on the robot in response to some internal logic in a ROS2 component. The response should be marshaled in a real-time manner. Real-time in this context is referring to the system's ability to both quickly and predictably produce the desired response from the robot. Including speed requirements in the term real-time is a normal misconception

of how real-time is defined, as real-time in no way denotes the quickness of a system, but rather its ability to reliably and predictably complete its real-time constrained task within the deadline defined for the task. However, real-time systems are oftentimes systems that are required to be able to *react quickly*. Nevertheless, the requirements of response quickness is not technically a real-time property. To avoid confusions and miscommunication, the quickness of the response will be separated as a separate property, referred to as the responsiveness.

4.1.1 Defining Key Performance Indicators

The joint control system is, in reality, composed of two control systems: the EGM control system inside the robot controller on the robot side and the ROS2 joint position controller on the ROS2 side. Having a networked distributed control system like this can introduce significant delays in the control of the robot. Having large delays between a motion is desired until it is executed can severely degrade the system's usefulness, and these should, therefore, somehow be measured

The developed ROS2 control system has a topic-based API towards the other nodes in the ROS2 side subsystem. This API can be used to communicate desired positions to the ROS2 joint position controller from another node. This other node can be written as a hardcoded node for performing a specified motion or a more complex subsystem of nodes forming an autonomous assembly application. Either way, in both scenarios, the node/application will need to communicate the joint goals to the joint position controller via a supported API. Likewise, for feedback, the node/application will need to subscribe to the joint states topic for feedback of status from the robot. Essentially, the application or node in question must interact with the joint position controller to command motion and interact with the joint state controller to receive feedback of state. Hence, it would be of interest to investigate the delay between a motion is commanded by a ROS2 application, and actually performed on the physical robot.

The motion delay would probably provide valuable information about the responsiveness of the control system. However, the pure latency is arguably also of interest, especially in regards to the real-time capacity of the system. Finding the worst-case round trip time (WCRTT) of a message between a ROS2 application and the state machine would give insight into the time requirements for transmitting a message from one end to the other. By measuring the WCRTT a great number of times, insights into the variability and mean of this Round Trip Time would be found. Worst case, mean and variability would together give valuable information about how hard real-time the system can handle and the transmission time between the state machine and a ROS2 application using the developed system to control YuMi. A WCRTT close to its mean RTT (MRTT) is desirable, as this indicates the system is able to deliver a semi-consistent RTT.

Continuing in the domain of real-time properties, both the end-to-end jitter and the packet loss over the Ethernet connection is of interest. End-to-end jitter refers to the deviation from the mean end-to-end delay. The jitter would provide information of the variability of the delay. In essence, jitter and latency variability refer to the same aspect of a data

transfer. An ideal connection has a constant delay, meaning zero jitter. Packet loss is of interest as packages not received cannot be acted upon. Measuring the number of packets lost would, therefore, be of interest.

Additionally, experiments must be conducted to investigate the control systems step response and general ability to follow a changing reference. The ROS2 integration of YuMi is at its core a integration of YuMi into an external control system, and the ROS2 joint position P-controller must be tuned to give satisfactory responsive control of the joints.

The following quantitative KPI's is then defined:

Defined KPI's
Little delay between motion is issued and completed
Little deviation between the WCRTT and the MRTT
Low end-to-end latency
Low end-to-end average jitter
Little packet loss over the Ethernet connection
Able to track a continuously changing reference fast and accurately

Table 4.1: The defined KPI's for evaluating the performance.

4.1.2 Determining Values Indicating Good Performance

having defined the KPI's in a quantitative manner, it is now necessary to define values constituting good performance in these seven dimensions. Whole works can, and are being, dedicated to determining realistic values for each property listed for different physical systems. This section will not provide empirically determined values representing the real system in a exact manner. The values will be suggested based on related work and correlation between the KPI's. The values are by no means exact, and is intended to form a hypothesis for what to expect and what can indicate the performance sought after in this chapter.

The first KPI is concerned with the motion delay. In [31] Ø. Mæhre was able to track a disk sliding down a ramp using EGM and an external vision system, lagging behind the disc with about a 200-300 millisecond delay. All though a ROS2 structure was not employed, the control structure is likely to be more computationally demanding than the control structure of the developed ROS2 system. The authors of [32] found round trip times between a ROS2 (Ardent) subscriber and publisher to be on the magnitude of between 1-3 milliseconds, with an average around the middle of the 1-3 ms range. This suggests that there is very little latency added by the ROS2 publish and subscribe data transfer and hence, it seems reasonable to expect at least similar ability to track a continuously changing reference.

The RTT described the time required to transmit a message to and from a end point. Having no motion to perform, the RTT should be significantly smaller than the motion

KPI	Suggested Value
Little delay between motion is issued and completed	Below 300ms
Little deviaiton between the WCRTT and the MRTT	Below 500%
Low end-to-end latency	Below 175 ms
Low end-to-end mean jitter	Below 45 ms
Little packet loss over the ethernet connection	Little expected.
Able to track a changing reference fast and accuratly	Able to track a reference within 0.1 degrees precision and a control lag of less then 300 ms.

Table 4.2: The defined KPI's with suggested limits.

delay. A rough estimation can be done by adding together known or partially known latency in the round trip. The round trip consists of a ROS2 round trip, a UDP round trip, and added delay from the added code in the state machine handling the communication. The EGM sensor protocol transmits messages at a 250 hz rate, adding only a 4ms latency. This latency will occur twice. The added latency by message handling on the robot side will likely not exceed 20 milliseconds. Adding up, an average RTT should be less than 50 milliseconds. A five times worse worst-case scenario would lead to very large deadlines to be needed for a round trip that often do not require so much time. Therefore a limit of 5*50 ms= 250 ms WCRTT limit is suggested. End-to-end latency being half the RTT, a limit of 175 ms is suggested. A low average jitter suggests stable performance, a desirable feature of a real-time system. An average jitter of less than 25% of the mean latency suggests fluctuations with an amplitude less than 50% mean latency, which intuition suggests to be conservative demand. The suggested maximum mean jitter is, therefore, 45 ms.

EGM sensor protocol uses UDP for transport, which normally is prone to packet loss. However, the combination of using a direct Ethernet connection and fixed transmission rate of 250 hz of which both sides are able to receive at, little packet loss is to be expected. Lower the packet loss, the better.

The ability to track fast and accurately a continuously changing reference is a combination of three properties: fast position control, robot accuracy and low motion delay. Fast control is a controller tuning issue, robot accuracy is non configurable and motion delay is actually considered by the first KPI. This KPI is therefore concerned with the robots practical ability to quickly and in real-time affect its environment. based on the suggested motion delay, and the accuracy of YuMi of 0.02 mm TCP accuracy, a control lag limit of 300 milliseconds to reach 0.1 degree accuracy is suggested.

Summarizing, the following KPI's and their suggested values, are presented:

4.1.3 Experiments to be Conducted

Having defined criteria that may indicate good performance, the next step is to define how these are to be measured. Though having defined seven KPI's this work is scoped to only conduct experiments for the last KPI concerned with tracking of continuously changing references. Measurements of the other KPI's will be discussed in "further work". The experimental setup will be reviewed lastly before turning to the results.

All experiments are performed on the right arm, on joint 6. Initial testing showed the response to be the same for both arms, and thus it was deemed unnecessary to perform the tests on both arms. **Joint Position Step Response**
Firstly, tuning of the ROS2 proportional gain will be conducted. This will be done using step-responses. The step response will be invoked from a separate ROS2 node. This node will subscribe to the /joint_states topic and find the initial position of the robot arm in question. Then a command with a 10 degree increment on one joint will be generated and published once. The state of the robot will be tracked continuously until it has reached the goal joint value within 0.1 degrees and held the position for some time.

Tracking a Sine Wave Reference
The continuously changing reference is chosen to be represented by a standing sine wave. It was chosen partly for its mathematical simplicity but also because it features both angle increments and decrements, in addition to three derivatives of position change. The reference will be sent from a separate ROS2 node. Likewise as in the step response test, the node will subscribe to the /joint_states topic and find the initial position of the robot arm in question. Then, in an while loop executing at 500 hz (twice the read/write frequency to the robot) the sine curve will be sent by sending the initial position, with increments added by eq. (4.1) listed below.

$$y(t) = A\sin(2\pi ft + \varphi) \tag{4.1}$$

4.2 Experimental Setup

The experiments will be conducted from a desktop PC connected to the robot via Ethernet. The robot runs the state machines, and the PC the ROS2 system and the node commanding a change in position. The PC will be idle, and will have no internet access, meaning the only network connection running will be the connection the robot. The PC used for the test have the following characteristics:

- Processor: AMD(R) Athlon(TM) ii x4 635 processor x4 (2.9 Ghz)
- RAM: 6 GB (533 Mhz)
- OS: Ubuntu 18.04.3 (Bionic Beaver)
- ROS2 version: Dashing
- Kernel version: 5.0.0-31-generic

- Link capacity: 100/1000 Mbps, Full-Duplex

EGM Settings and Dynamic Load Error

An error kept occurring during the testing of the architecture. The error was occurring on the robot side, in the robot controller. The error in question was the 50375, *Dynamic load too high*. The error message states the error to be caused by a required torque for one of the robot axes to be too high. The error occurs when an arm is asked to position several joints at once, occurring more frequently for larger movements than small movements. When the error occurs, the motors and the execution of the state machine are immediately stopped. The available manuals for YuMi and the irc5 robot controller were consulted, but no solution was found. Internet and forum searches provided some more insight into the issue. Several users online found the error to connected to joint acceleration and found that by limiting the Max Speed Deviation EGM parameter to remove the issue. Limiting Max Speed Deviation to $10deg/s^2$ removed the issue, however having so severely limited the allowed acceleration, the motion was now extremely slow. As the step response and sine wave tracking did not provoke the error, the tests have been conducted using both the reduced speed EGM settings and the fullspeed EGM settings.

The fullspeed EGM setttings are listed in table 4.3. Max Speed Deviation is set according to the lowest allowed maximum speed found for the joints, found to be $180deg/s$. This is in accordance to the recommendation found in the state machine RAPID code. The reduced speed settings, shown in table 4.4, are identical to the fullspeed with the exception of Max Speed Deviation. Only run time configurable EGM settings were modified, i.e. only the EGM settings configurable in the RAPID code divert from the default. Settings not mentioned, no matter type, are left to default. EGM system settings were left to default.

EGM setting	Value	Description
Use Filtering	False	Layer 1 filtering
Comm timeout	60	Communication timeout {s}
Low Pass filter	0	Low Pass filter bandwidth [hz]
Sample rate	4	Sample rate of the EGM communicating [ms]
Max Speed Deviation	180	Maximum admitted joint speed change [deg/s]
Ramp in time	0.1	Ramp in time [s]
Position Correction Gain (PCG)	1	Position correction gain of the EGM controller
Ramp in time	0.1	Ramp out time [s]

Table 4.3: EGM runtime settings, fullspeed mode.

EGM setting	Value	Description
Use Filtering	False	Layer 1 filtering
Comm timeout	60	Communication timeout {s]
Low Pass filter	0	Low Pass filter bandwidth [hz]
Sample rate	4	Sample rate of the EGM communicating [ms]
Max Speed Deviation	10	Maximum admitted joint speed change [deg/s]
Ramp in time	0.1	Ramp in time [s]
Position Correction Gain (PCG)	1	Position correction gain of the EGM controller
Ramp in time	0.1	Ramp out time [s]

Table 4.4: EGM runtime settings, reduced speed mode.

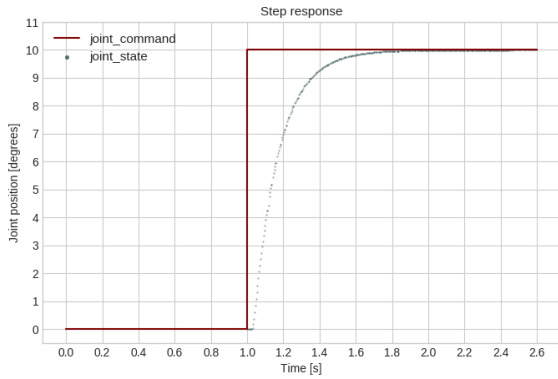
Chapter 5

Results

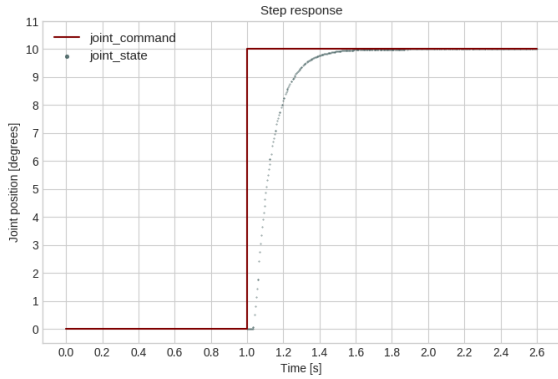
This chapter will present the results obtained from the conducted experiments. Results obtained with both the fullspeed EGM settings and the reduced speed settings will be presented. Results will be discussed in chapter 6.

5.1 Step response

Fullspeed Mode
The step response tests for two different proportional gains (P) in fullspeed mode are presented below in fig. 5.1a) for P=1.2, and in fig. 5.1b) for P=1.6. The step is completed in 0.8 seconds with P=1.2, and 0.6 seconds with P=1.6. P=1.6 thus, unsurprisingly, provides the fastest response, completing the step in about 200 ms faster than P=1.2. By further testing it was determined that oscillations starts to occur at around P=2, indicating that proportional gain values higher than P will give a underdamped or, worst case, and unstable response.



(a) P = 1.2

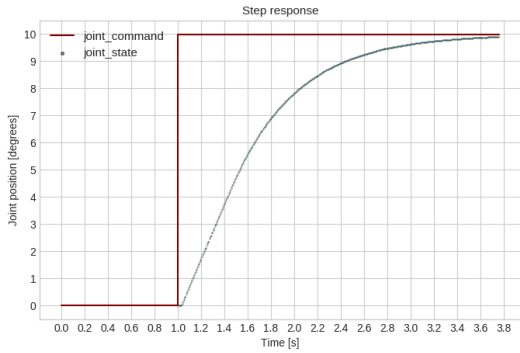


(b) P = 1.6

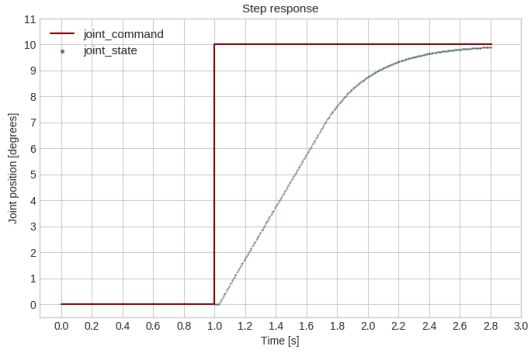
Figure 5.1: Step responses, fullspeed settings.

Reduced speed mode

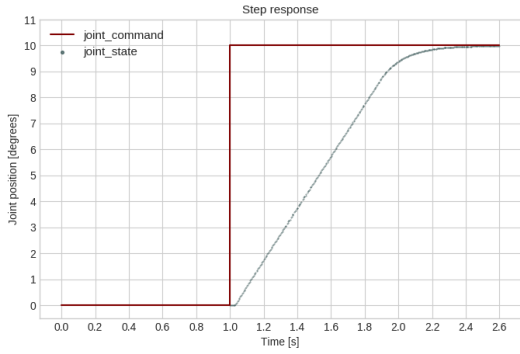
The step response tests was conducted also for the reduced speed settings. Due to higher speeds being unavailable with the the settings, lower proportional gains were used. The results are seen in fig. 5.2 a) for $P = 0.35$, fig. 5.2 b) for $P = 0.6$ and fig. 5.2 c) for $P = 1.24$. The throttled maximum allowed acceleration can be seen in the linear regions of the responses. Predictably, $P = 1.2$ provides the fastest response, completing the step in about 1.4 seconds. When further increasing P, the completion time will approach 1 second, however not reaching it due to the ramp in and ramp out.



(a) P = 0.35



(b) P = 0.6



(c) P = 1.2

Figure 5.2: Step responses, reduced speed settings.

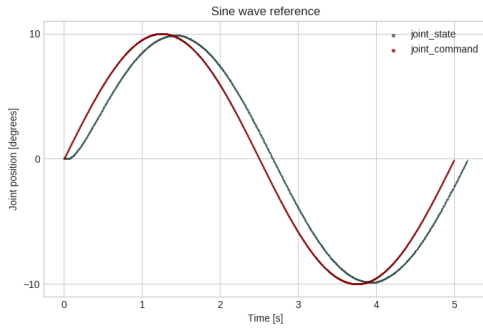
No significant steady-state error were observed during the testing.

5.2 Sine Wave Tracking

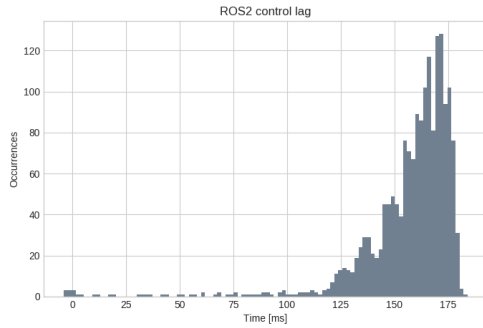
Fullspeed Mode
The results from the sine wave reference tracking for two different proportional gains are presented in fig. 5.3 for $P = 1.2$ and in **??** for $P = 2$. Both responses track the reference well, reaching all the commanded positions, never lagging more than 184 ms behind the reference. With the proportional gain P set to 1.2, the mean delay is found the be around 156 ms and the variability is found to be rather small, as shown in the histogram in fig. 5.3b. By increasing the proportional gain, a closer tracking can be achieved. With a more aggressive controller, this is expected. When increasing the proportional gain to 2, the mean delay drops to about 92 ms and the maximum drops to about 106 ms. The variability also drops, as visible in the quite narrow histogram in fig. 5.4b However, due to the oscillations starting at this gain level, the gain should not be increased further.

Control delay	
Max delay [ms]	184
Mean delay [ms]	156

Table 5.1: Sine wave reference, fullspeed settings, P = 1.2



(a) Reference tracking

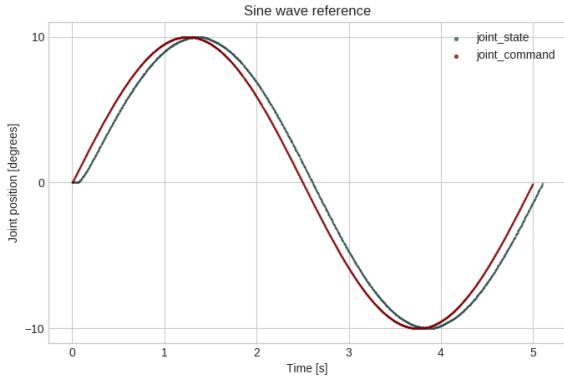


(b) Control lag

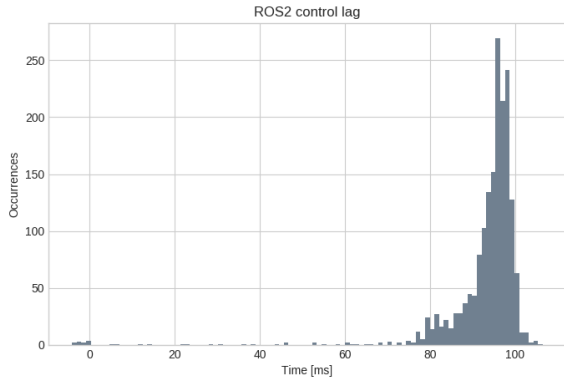
Figure 5.3: Control delays during tracking of reference, P = 1.2

Control delay	
Max delay [ms]	106
Mean delay [ms]	92

Table 5.2: Control delays during tracking of reference, P = 2



(a) Reference tracking

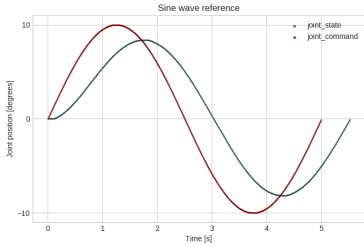


(b) Control lag

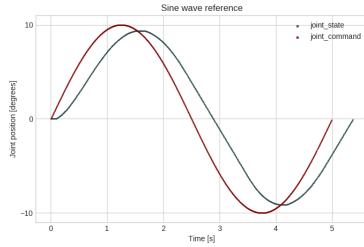
Figure 5.4: Sine wave reference tracking using rfullspeed settings.

Reduced speed Mode

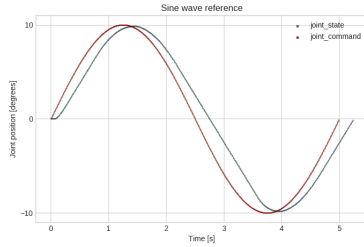
The results from tracking of the sine wave reference in reduced speed mode for three different proportional gains are presented in fig. 5.5a for $P = 0.35$, in fig. 5.5b for $P = 0.6$ and in fig. 5.5c for $P = 1.2$. The figures show the reference to be tracked increasingly well with increasingly higher proportional gain, confirming the trend observed earlier. For the lower proportional gains, the reference is not tracked accurately. With $P = 1.2$ however, the joint is able to track the reference, however with a rather large control lag. The control lag seems also to vary quite a bit, which is confirmed by consulting the histogram in fig. 5.6.



(a) Reference tracking, P = 0.35



(b) Reference tracking, P = 0.6



(c) Reference tracking, P = 1.2

Figure 5.5: Sine wave reference tracking using reduced speed settings.

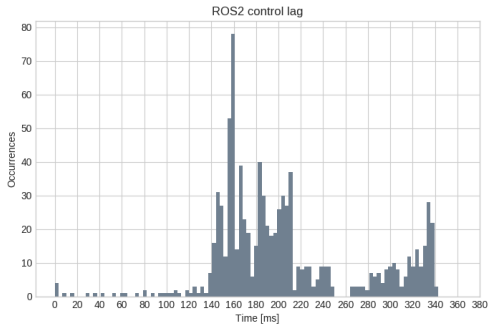


Figure 5.6: Control lag.

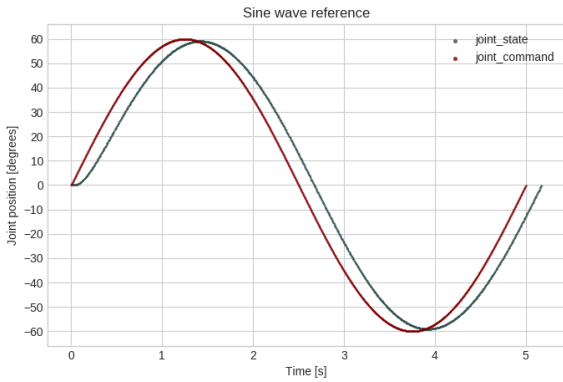
Fullspeed Mode - Bigger Motion

An additional fullspeed tracking was conducted. In this test the amplitude of the sine wave was increase from 10 degrees to 60 degrees. The intention of this test is to evaluate weather the tracking can track as closely for larger reference changes. The results from the tracking is shown in fig. 5.7. From fig. 5.7 the reference can be seen to track similarly as the 10 degree amplitude test. Furthermore the variability do not seem to have increased, witch is also indicated by the histogram, shown in table 5.3. However, the number of matched

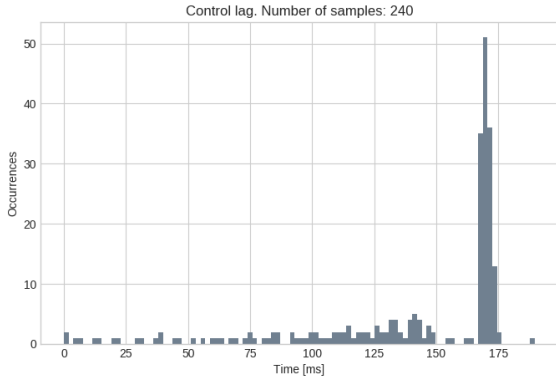
samples have dropped, indicating the matching algorithm cannot match as many points in accordance to the 0.1 degree proximity requirement as earlier.

Control delay	
Max delay [ms]	141
Mean delay [ms]	189

Table 5.3: Control delays during tracking of reference, P = 1.2



(a) Reference tracking



(b) Control lag

Figure 5.7: Sine wave reference tracking using fullspeed settings.

Chapter 6

Discussion

6.1 Dynamic load too high

The occurrence of *Dynamic load too high* errors seems to be coupled to the allowed joint angular acceleration allowed. Limiting the EGM runtime parameter Max Speed Deviation to 10 deg/s^2 seems to solve the issue, however, the speed at this setting is extremely slow. On a normal standalone irc5 controller, there is manual reduced speed mode with a speed limitation of 250mm/s. It is possible that there resides an undiscovered safety mode or feature on YuMi's integrated irc5 controller, causing the error. All external control is done with the robot controller in Auto mode, a mode typically not throttled with a speed restriction. Therefore, while it may be a safety feature residing somewhere blocking high-speed movements in Auto mode, it is in the author's opinion that it is far more likely that the cause is a system setting somewhere. During the later stages of the development process, during the documentation phase, a discovery was made related to the $k - factor$ in the EGM control system. While the positional correction gain has been managed through RAPID code, the system configuration proportional position gain (PPG) has not been modified from its default value. Remembering that the $k - factor$ amplifies the commands received from the EGM server and that the $k - factor$ is defined as $k = PCG * PPG$, this should be looked further into. However, until the issue is fixed, YuMi is deployed with reduced speed settings.

6.2 Step Response

Lots of the tracking properties presented can be explained by a fast step response. In fullspeed mode, $P = 2$ strongly suggests good tracking properties when operated at this proportional gain. This was also seen in the tests. As oscillations started occurring beyond $P = 2$, it is in practice the gain limit for which performance is not gained by increasing it further. It has been assumed throughout that a fast step response is desirable for YuMi. The reason being that even if deterministic execution times and low latency control were

achieved, slow control will render the motion too slow to affect the environment in the desired real-time manner. However, it is not assumed that fast control can remove the need for deterministic properties of the system nor remove non-control-related latency from the system.

6.3 Sine Wave Tracking

As mentioned, it has been speculated throughout that a fast position control in tandem with low latency control cycles will be able to provide YuMi with a control system able to affect its environment in real-time, or near-real-time. The sine wave tracking tests have in some aspects, been a test of the systems ability to do just that. The tests showed that a fast joint position controller will reduce the delay between the desired position being commanded, until it is reached. Using the fullspeed settings, the tests found the control lag to be fairly consistent and of a manageable scale. When using a conservative proportional gain of $P = 1.2$ it was found that for both large sine waves with a range of 120 degrees and smaller with a range of 20 degrees, the robot would lag about 150 ms behind the desired position quite consistently.

A drawback of this test is that it is hard to differentiate between how much of the 150 ms lag is caused by system latency, and how much is due to the controller's update law. As a result, little is actually known at the moment about the latency characteristics of the system. However, knowledge has been gained in the system's ability to affect its environment in real-time, which has been assumed by the author to be almost equally important. Therefore, when concentrating on a single test, these tests were assumed to be the most critical.

With the reduced speed settings, the controller lag is unsurprisingly quite a bit bigger. Comparing the $P = 1.2$ tests conducted using both sets of settings, it is clear that even with the same proportional gain, the reduced speed controller will lag more behind. When consulting the step responses for $P = 1.2$ for both sets of settings, this is not too surprising. Perhaps more surprising is the severely increased variability caused by the acceleration throttling. This could perhaps be explained by throttled acceleration rendering the robot unable to perform bigger steps when necessary.

6.4 Latency Characteristics of the Architecture

Removing latency, especially unpredictable latency, is important when designing real-time software. When the control architecture was developed for YuMi this was considered when making components interacting with the EGM control loop being run by AbbEgmHardware's executable. The decision to use ros2_controls as the control framework on top of which the ROS2 application could be built was partly due to this. Also, one of the main reasons for choosing Externally Guided Motion as the interface to the robot controller was its high frequency, low latency streaming communication. However, little optimization for low latency has been performed on the codebase. Specifically, can probably significant

latency be shaved off from joint_position_controller's update() method. Considering it is called in the middle of the latency-sensitive EGM control loop, it contains a significant amount of logic for updating the command. At the moment, it uses a PID-controller with the gains for its I and D terms set to zero to make it in practice a P-controller. This is very computationally wasteful as the calculations of the terms are still being performed, even though they always are found to give zero contribution. It has been assumed throughout that the ros2_control framework is suitable for real-time control, however, no tests have been performed on the framework to determine whether this is a correct assumption. Ad hoc inspection of the code has, in the meantime, given an impression that is is. This impression is mainly based in that the framework composes the components involved in the control loop into a single process, avoiding unnecessary process switching, and the passing of pointers to stationary data. Compared to other solutions, like copying and get() method calls, this a low latency solution.

6.5 Towards a Intuitive Architecture

Besides being made to handle real-time control, an additional goal was to implement the architecture in an intuitive fashion with a well-defined API. This goal was assumed to be reached if three sub-goals were reached: a well-defined API, components are designed to be cohesive, and to encapsulate a single functionality or role and for the architecture to be re-configurable without recompilation needed. A well-defined API was partly reached. While the architecture has provided some ease-of-use, its gripper API very constrained and suboptimal. Besides only supporting grip movements, its API is not defined in an intuitive manner.

In the theme of the digital twin from the Industry 4.0 landscape, components could be designed to form virtual representations of the hardware it interfaces with. Being quite a lofty goal, and out of the scope of this work, it was decided upon cohesive components encapsulating an entity's functionality instead. Meaning e.g., that each arm must have its own hardware interface instance, as no dual-arm entity exists physically on the robot, but only one robot_manager, as there is only of the robot. And, likewise, for the grippers, there must be two Smart gripper instances representing one unique gripper each. From this design principle, a need for configurable classes surfaced. It was early decided upon to use config files and parameter servers to make the information needed to instantiate an instance available to the ROS2 namespace. When implemented, it showed to be an effective data bank for the namespace.

Chapter 7

Conclusion and Further work

7.1 Concluding the Work

Small parts assembly puts requirements on the dexterity and maneuverability of the robot manipulator. To facilitate small part assembly, dual-arm robots like ABB IRB 14000 YuMi is developed. Featuring 7 joint on each arm, padding, and safety mechanisms for safe operation side-by-side human workers, this type of industrial robot presents a shift in industrial robotics. The new collaborative industrial robots introduce new possibilities for flexible robotics. This, however, requires new development strategies to be employed. When implemented in a dynamically changing environment, the ability to react to the environment in a timely manner also becomes of importance. Developing robot software is challenging, especially with real-time requirements. However, with the advent of ROS2 this is made easier. ROS2 featuring DDS and Quality of Service options enables real-time systems to be developed on ROS2.

In this report, integration of the ABB IRB 14000 YuMi with ROS2 is presented. Using the Externally Guided Motion (EGM) interface from ABB, a ROS2 control system is able to control YuMi to track a continuously changing references in real-time. Experiments conducted found most configurations to present a mean control lag under 200 ms, with the best results reaching latency below 100 ms. However, an acceleration throttler is found somewhere in the system, requiring a manual throttling of the maximum allowed acceleration to avoid system stops.

7.2 Further work

joint_velocity_controller
The current architecture utilises a joint_position_controller exclusively to control the joints of the robot. Controlling only the position, the speed of the motion is not controllable from the controller. The speed is at the moment only possible to configure using throttling of

acceleration on the robot controller, with Max Speed Deviation, or by adjusting the proportional gain on the P-controller of the joint_position_controller. Being able to control the velocity of the motion could be of great use if YuMi will need affect a dynamically changing environment i real-time. It would allow the ROS2 application move the joints faster when manipulator is far from its target and perform a controlled deceleration when reaching its target, thus achieving potentially faster and smoother control than that of a joint_position_controller.

Measuring Latency's in Proper To chart Real-Time Characteristics
It would be useful to measure some response times and end-to-end latency's to chart the system real-time characteristics. The conducted tests, while useful, provide little knowledge of execution time and latency's in the system. To form a profile on the real-time properties of the system, a series of end-to-end latency tests, Round Trip Times and jitter measurements should be performed. With information about the kind of deadlines the components can manage, more interesting applications can be formed.

Bibliography

[1] B. Siciliano, L. Sciavicco, L. Villani, and G. Oriolo, *Robotics: modelling, planning and control*. Springer Science & Business Media, 2010.

[2] K. M. Lynch and F. C. Park, *Modern Robotics*. Cambridge University Press, 2017.

[3] R. S. Hartenberg and J. Denavit, "A kinematic notation for lower pair mechanisms based on matrices," *Journal of applied mechanics*, vol. 77, no. 2, pp. 215–221, 1955.

[4] J. G. Balchen, T. Andresen, and B. A. Foss, *Reguleringsteknikk*. NTNU, Institutt for teknisk kybernetikk, 2004.

[5] N. S. Nise, *CONTROL SYSTEMS ENGINEERING, (With CD)*. John Wiley & Sons, 2007.

[6] *Ros introduction*, `http://wiki.ros.org/ROS/Introduction`, 2019.

[7] *Ros distributions*, `http://wiki.ros.org/Distributions`, 2019.

[8] *Ros wiki*, `http://wiki.ros.org/`, 2019.

[9] P. A. Laplante, "Real-time systems design and analysis," Wiley Online Library, 2004.

[10] D. Wetteroth, *OSI reference model for telecommunications*. McGraw-Hill Professional, 2001.

[11] Google, *Protocol buffers - developers guide*, `https://developers.google.com/protocol-buffers/docs/overview`, [Online; accessed 22-November-2019].

[12] *Application manual - controller software irc5*, 3HAC050798-001, Rev. C, ABB AB, Robotics, 2016.

[13] ABB AB, Robotics, *Abb unveils the future of human-robot collaboration: Yumi®*, `https://new.abb.com/news/detail/13110/abb-unveils-the-future-of-human-robot-collaboration-yumir`, [Online; accessed 1-December-2019], 2014.

[14] *Yumi - press release backgrounder*, ABB AB, Robotics, Jan. 2016.

[15] ——, *Yumi® - irb 14000 — collaborative robot*, https://new.abb.com/products/robotics/industrial-robots/irb-14000-yumi, [Online; accessed 1-December-2019], 2019.

[16] A. Robotics, "Brochure: Yumi® creating an automated future together," *You and me*, vol. 14, p. 2017, 2017.

[17] *Product specification - controller irc5*, 3HAC047400-001, Rev. U, ABB AB, Robotics, 2019.

[18] *Product specification - irb 14000*, 3HAC052982-001, Rev. J, ABB AB, Robotics, 2018.

[19] *Operating manual robotstudio*, HAC16590-1, Rev. K, ABB ABB, Robotics, 2007.

[20] *Operating manual - robotstudio*, 3HAC032104-001, Rev. L, ABB ABB, Robotics, 2013.

[21] ABB AB, Robotics, *Ease-of-use packages between ros and abb robots*, URL: https://www.youtube.com/watch?v=MunLpU5SZlc&t=423s, Dec. 2018.

[22] *Robot web services*, http://developercenter.robotstudio.com/webservice/api_reference, [Online; accessed 20-November-2019], 2019.

[23] *Robotware add-in - user manual - statemachine add-in 1.1*, ABB AB, Robotics, 2019.

[24] R. ABB AB, *Statemachine add-in*, https://robotapps.robotstudio.com/#/viewApp/7fa7065f-457f-47ce-98d7-c04882e703ee, [Online; accessed 12-November-2019], 2018.

[25] ABB, *Abb_librws*, https://github.com/ros-industrial/abb_librws, 2019.

[26] ——, *Abb_libegm*, https://github.com/ros-industrial/abb_libegm, 2019.

[27] *Ros-controls*, https://github.com/ros-controls, 2019.

[28] S. Chitta, E. Marder-Eppstein, W. Meeussen, V. Pradeep, A. Rodríguez Tsouroukdissian, J. Bohren, D. Coleman, B. Magyar, G. Raiola, M. Lüdtke, and E. Fernández Perdomo, "Ros_control: A generic and simple control framework for ros," *The Journal of Open Source Software*, 2017. DOI: 10.21105/joss.00456. [Online]. Available: http://www.theoj.org/joss-papers/joss.00456/10.21105.joss.00456.pdf.

[29] *Abb_yumi*, https://github.com/Mariunil/abb_yumi, 2019.

[30] *Ros2_controllers*, https://github.com/ros-controls/ros2_controllers, 2019.

[31] Ø. Mæhre, "Following moving objects using externally guided motion (egm)," Master's thesis, UiS, Jun. 2016.

[32] C. S. V. Gutiérrez, L. U. S. Juan, I. Z. Ugarte, and V. M. Vilches, "Towards a distributed and real-time framework for robots: Evaluation of ros 2.0 communications for real-time robotic applications," *arXiv preprint arXiv:1809.02595*, 2018.

[33] *Product manual - grippers for irb 14000*, 3HAC054949-001, Rev. H, ABB AB, Robotics, 2018.

Appendix

7.3 Auxiliary equipment

7.3.1 Smart Gripper data and specifications

Servo module specifications

Description	Data
Travel length	0-50 mm (max- 25 mm per finger)
Maximum speed	25 mm/s
Repeatability	$\pm$ 0.05 mm
Gripping direction	Inward or outward
Maximum gripping force	20 N (at the gripping point of 40 mm)
External force (not in gripping direction)	15 N (at the gripping point of 40 mm)
Force control accuracy	$\pm$ 3 N

Table 7.1: Servo module specifications. Data from [33]

Vision module specifications

Description	Data
Resolution	1280x1024
Lens	6.2 mm f/5
Illumination	Integrated LED with programmable intensity

Table 7.2: Vision module specifications. Data from [33]

7.4 ROS2 node lifcycle

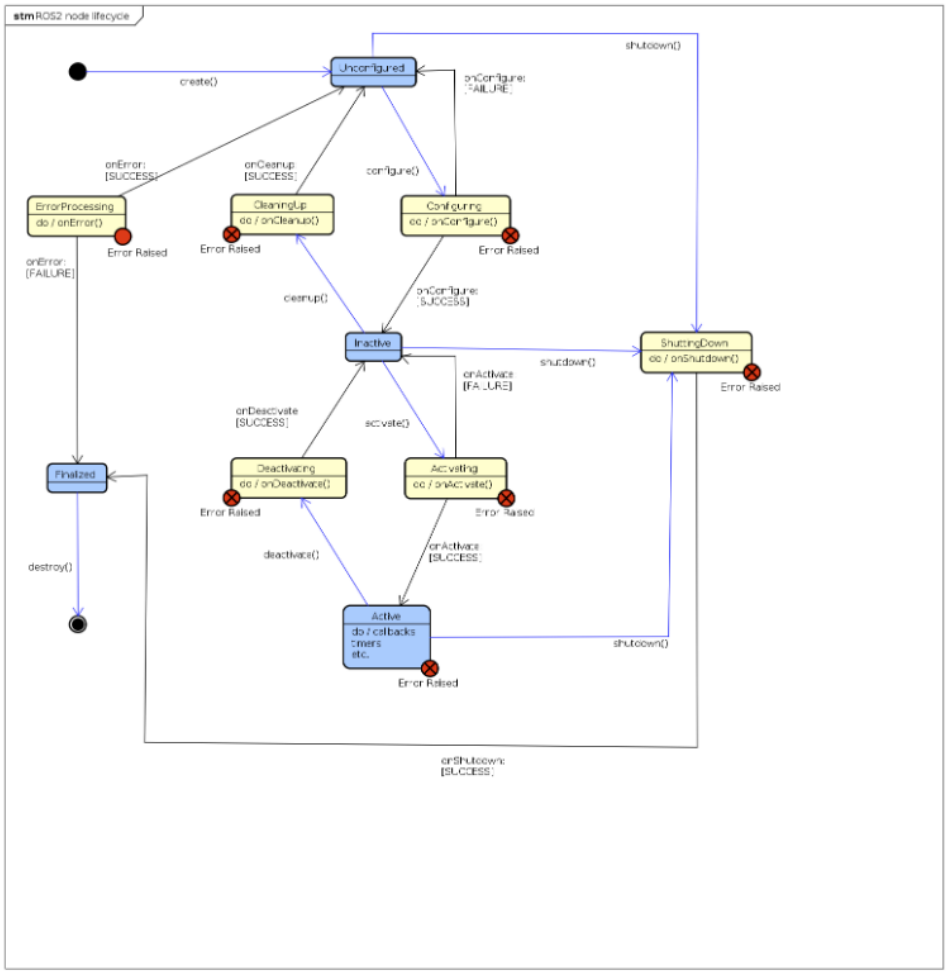


Figure 7.1: The ROS2 managed lifecycle.