Tampereen yliopisto

Jaakko Hautamäki

INTERFACING EXTENDED REALITY AND ROBOTIC OPERATING SYSTEM 2

Master Thesis
Faculty of Information Technology and Communication Sciences
Professor Timo Hämäläinen
Doctoral Researcher Matti Käyrä
August 2021

ABSTRACT

Jaakko Hautamäki: Extended reality and robotic operating system 2
Master's Thesis
Tampere University
Master's Program in Information Technology
August 2021

Extended reality and robotic operating system 2 are technologies in an evolving state that have recently become popular. The development of both technologies has been rapid in the recent years and supported by multiple powerful companies. To adapt these technologies, in research and product development, their state-of-the-art should be studied and outlined regularly. This allows for more educated choices in planning of products and research targets, on the account of the technologies and the features within.

The goals of the project were to document the state-of-the-art of extended reality and robotic operating system 2 as well as to identify any future research targets within those technologies. In addition, an important goal was to try to create an interface between these technologies and evaluate its features as well as present and future potential. The interface was tested and expanded to other platforms as extensively as possible in the available time frame.

A comprehensive state-of-the-art of both technologies was established and an interface was created between them using the Unity game engine. For the creation of the interface, applicable software and software interfaces were researched. The creation and development process of the interface and the problems discovered were documented. The usability, robustness and expandability of the interface were evaluated, within the boundaries of the project schedule, through unit testing, integration testing, stress testing and feedback from demonstrations of the interface.The first demonstration covered the functionality and features of the interface from a robot into Unity, while the second demonstration presented the possibilities of the interface in conjunction with extended reality features.

The project was a successful in researching and documenting the state-of-the-art and creating an interface between the two technologies, which received positive feedback from the demonstrations. The interface software ran on Unity and could successfully send and receive different message types to and from ROS2 environment. The second demonstration also gave a good example of how the combination of ROS2 and XR could be used in practicality. The interface was successfully expanded into robotic operating system 1 environments with ROS1_bridge software. Significant latency issues were discovered during the use of ROS1_bridge however. The issues were researched and documented. The interface was successfully compiled for Windows 10 and Ubuntu Linux environments, but attempts at compiling for Android failed. Multiple future research targets were discovered from the study as well.

Keywords: Robotic operating system 2, ROS 2, Extended reality, XR, Virtual reality, VR, Augmented reality, AR

TIIVISTELMÄ

Jaakko Hautamäki: Laajennettu todellisuus ja robotic operating system 2
Diplomityö
Tampereen yliopisto
Tietotekniikan DI-Ohjelma
Elokuu 2021

Laajennettu todellisuus (eng. Extended reality) ja robotic operating system 2 ovat murrosvaiheessa olevia teknologioita, jotka ovat yleistymässä. Molempien teknologioiden kehitys on ottanut suuria edistysaskeleita viime vuosina ja monet suuren mittaluokan yhtiöt tukevat niiden kehitystä. Jotta näitä teknologioita voidaan ottaa käyttöön niin tutkimus kuin tuotekehitysmielessä, tulee niiden kehityksen tilannetta tutkia ja dokumentoida säännöllisesti. Teknologian tilan tunteminen mahdollistaa valistuneempia valintoja tuotteen tai tutkimuskohteen suunnittelussa teknologian ja sen ominaisuuksien osalta.

Työn tavoitteena oli kartoittaa laajennetun todellisuuden ja robotic operating system 2:sen nykytilan lisäksi keskeisiä tulevaisuuden tutkimuskohteita. Tämän lisäksi olennainen tavoite oli tutkia, olisiko mahdollista luoda rajapintaohjelmisto näiden teknologioiden välille ja minkälaisia ominaisuuksia ja mahdollisuuksia sillä voisi saavuttaa nyt ja tulevaisuudessa. Rajapintaohjelmisto testattiin ja sitä yritettiin laajentaa useille alustoille aikataulun puitteissa.

Työssä muodostettiin kattava tilannekatsaus molemmista teknologioista ja luotiin niiden välille ohjelmistorajapinta käyttäen Unity-pelimoottoria. Rajapinnan luontia varten tutkittiin käyttötarkoitukseen soveltuvia ohjelmistoja ja ohjelmistorajapintoja. Rajapinnan luonti- ja kehitysprosessi sekä löydetyt ongelmat dokumentoitiin. Muodostetun rajapinnan käytettävyyttä, toimintavarmuutta ja laajennettavuutta arvioitiin aikataulun puitteissa yksikkötestauksella, integraatiotestauksella, stressitestauksella sekä palautteella, jota saatiin rajapinnan demonstraatio-esityksistä. Ensimmäinen Demonstraatio-esitys käsitteli itse rajapinnan toimintaa sekä ominaisuuksia robotista Unityyn ja toinen esitteli rajapinnan mahdollisuuksia yhdistettynä lisätyn todellisuuden ominaisuuksiin.

Projektissa onnistuttiin tutkimaan ja dokumentoimaan ROS2:sen ja lisätyn todellisuuden nykytila sekä luomaan rajapinta näiden välille, joka sai myös positiivista palautetta demonstraatioista. Rajapintaohjelmisto toimi Unity:ssä ja pystyi lähettämään ja vastaanottamaan eri viestityyppejä ROS2-ympäristöstä. Toinen demonstraatio näytti myös onnistuneesti miten ROS2:sen ja lisätyn todellisuuden yhdistelmää voidaan hyödyntää käytännössä. Rajapinta saatiin onnistuneesti laajennettua robotic operating system 1 -ympäristöön ROS1_bridge-ohjelmiston avulla. ROS1_bridgen käytössä tosin huomattiin latenssi-ongelmia, jotka tutkittiin ja dokumentoitiin. Rajapinta käännettiin onnistuneesti Windows 10 ja Ubuntu Linux -ympäristöihin, mutta yritykset kääntää rajapintaa Androidille epäonnistuivat. Teknologioista löydettiin myös useampia tulevaisuuden tutkimusaiheita.

Avainsanat: Robotic operating system 2, ROS 2, Laajennettu todellisuus, XR, Virtuaalitodellisuus, VR, Lisätty todellisuus, AR

PREFACE - ALKUSANAT

Tahtoisin kiittää VTT:n työntekijöitä avusta ja hyvästä työympäristöstä diplomityön aikana, erityisesti Kaj Heliniä, Taru Hakasta, Jaakko Karjalaista, Vladimir Goriachevia ja Petri Tikkaa. Teidän ohjauksenne ja kommenttinne tekivät työstä viimeisen päälle hiotun. Kiitokset myös Timo Hämäläiselle työn ohjauksesta ja Matti Käyrälle työn tarkastamisesta. Teiltä sai hyviä korjausehdotuksia ja tukea työhön.

Kiitokset myös kommenteista, jatkuvasta tuesta ja tsemppauksesta Rina Hautamäelle ja Anja Studerille. Teidän takia sitä jaksaa pitää päänsä pystyssä. Viimeisenä, suuret kiitokset Jouko Hautamäelle elämän eväistä läpi vuosien, jotka ovat vieneet näinkin pitkälle. Pojastasi tuli diplomi-insinööri.

Tampereella 23.08.2021

Jaakko Hautamäki

INDEX

ABREVIATIONS AND CONCEPTS

API	Application interface
AR	Augmented reality
CAVE	CAVE automatic virtual environments
cobot	Collaborative robot
DDS	Data distributed service
DLL	Dynamic link library
DOF	Degrees of freedom
EEG	Electroencephalography
EOL	End of life
FOV	Field of view
Haptic	“Digital Technology of or relating to tactile sensations and the sense of touch as a method of interacting with computers and electronic devices” [1]
HMD	Head-mounted display
JSON	Javascript object notation
LAN	Local area network
LIDAR	Light detection and ranging
macro	A keyboard shortcut or a small program or script used to automate common tasks [2]
MR	Mixed reality
NDA	Non-disclosure agreement
QoS	Quality of service
RCL	ROS client support library
ROS	Robotic operating system
ROS2-Unity library	The custom software library developed during this thesis project
(screen) burn-in	A residual image left on a screen after displaying the same image for a long time [2]
SLAM	Simultaneous localization and mapping
struct	A struct (short for structure) is a user-defined data type that can store multiple related items and it is available in C based programming languages [2]
subpixel	Any of the units that make up a pixel. Each pixel usually has one red, one blue, and one green subpixel [3]
telepresense	“The use of virtual reality technology to operate machinery by remote control or to create the effect of being at a different or imaginary location” [1]
UR	Universal robotics
VR	Virtual reality
VTT	Technical research centre of Finland (finnish. Valtion teknillinen tutkimuskeskus)
WIFI	“A system for connecting electronic equipment such as computers and electronic organizers to the internet without using wires” [4]
WMR	Windows mixed reality
XR	Extended reality

1. INTRODUCTION

The thesis was done for the Technical Research Centre of Finland (VTT) and the aim was to study and outline the state-of-the-art of Robotic Operating System 2 (ROS2) and Extended Reality (XR). Additionally, the aim was to build an interface between them using Unity game engine, assuming it was feasible, practical and reusable. Some research has already been done on this field and also some software are available that might be applicable for this purpose. The features of these software were tried and evaluated as part of this project. In addition, any significant problems or features discovered during the research and development of the interface were evaluated and documented.

XR is not a new idea, but it has become exceedingly more popular and diverse during the last decade, especially during the last few years, with the introduction and evolution of head mounted XR displays. XR has also evolved from the gaming and entertainment sector into the industry. With the XR technology rapidly evolving, there is a need to research and document its current status from time to time. This allows more educated visioning and planning of future utilizations of XR as well as avoiding the pitfalls of outdated technology and methods.

ROS is in the process of a major evolution step as well as it is moving from the initial ROS1 to ROS2, which overhauls the whole system rather extensively. The evolution has not been completely realized yet however. ROS1 already has a large userbase with wide array of software packages and translating these packages to ROS2 might not be quick nor trivial. Thus research is required to document the current status of ROS2 and the evolution from ROS1 to ROS2 in order to find any pitfalls in the evolution and ROS2 in general. Also same as with XR, research allows for more educated visioning and planning of future utilizations of ROS2. As a distinction, in this thesis the term ROS1 is used to refer to the first generation of ROS, as opposed to ROS2, while ROS is used as a term for the whole ROS ecosystem, consisting of both ROS1 and ROS2.

Unity is one of the most popular game engines available. It supports wide array of different platforms and has also supported XR development for years at the time of writing [5, 6]. Unity technologies have also made a ROS plugin for Unity, with ROS2 integration in alpha testing stage [7]. This was briefly evaluated as part of this thesis. Unity was chosen as an essential tool for this thesis by the request of the employer. [8, 9]

One practical goal for the state-of-the-art and the interface would be to advance and act as a baseline for future research projects in these fields. As ROS2 and XR are hypothesized to become widely used in the future, they are fields that should be researched. To do so, a baseline overview should prove valuable for anyone wishing to dive deeper into the fields. However, the core part of the thesis was to try and combine these two fields with an interface, to see if it is possible and what its capabilities are. Within the boundaries of the schedule, the expansion of the interface to different platforms, such as Android was tried. Furthermore, the interface was tested for faults as possible. The software API created was named ROS2-Unity library.

The future use cases for such combination could include surveillance or safety critical work with remote control of robots with strong telepresence through XR or monitoring of varied data from ROS2 network in an XR environment for instance.

The state-of-the-art research is done with qualitative research through analyzing news articles about device and technology features and measuring them against and various application interface (API) documentations and scientific research papers applicable. As both technologies are in a state of rapid evolvement, up-to-date scientific research on them is scarce.

With the rapid development of ROS2, the documentation webpage of ROS2 seems to be out-of-date and sparse in some parts, especially when it comes to status of the porting of tools and their features from ROS1 to ROS2 [10]. For this reason, some of the information about ROS2 is devised from ROS1 documentation along with indirect deduction from other sources and may be uncertain in nature.

The research of the available applicable software, as well as design and implementation of the interface, was done through researching the various API documentation available and through trial and error. As the documentation of new or rapidly evolving technologies often falls behind in the development process, it was necessary to sometimes read source code and try out different functions to successfully utilize or compile the software API.

The implementation of the custom API and any problems or notable features are evaluated through mixed method, interpretative research approach, as the interface is designed based on the features discovered in the state-of-the-art review as well as the available applicable software or ROS2 API documentation and source code. The methods are picked according to the problem or feature. In addition, demonstrations of the capabilities of the interface are given and the interface is evaluated according to the

feedback received during and after the demonstrations. The feedback is given in an unstructured way.

The thesis is structured in the following way. Chapters 2 and 3 describe the state-of-the-art of ROS2 and XR respectively. Chapter 4 covers the available software that could be applied for interfacing ROS2 with Unity and XR and evaluates them. Chapter 5 covers the final design of the implemented custom API in high abstraction level, while chapter 6 describes the API software and its development process in more practical detail, including the testing of the software during development. Chapter 7 covers the demonstration of the developed API and Chapter 8 summarizes the project in the light of the planned goals in the beginning of the project.

2. ROBOTIC OPERATING SYSTEM

The Robotic Operating system, unlike its name would apply, is not an operating system, like Linux or Windows for instance. It is an open source middleware or software development kit (SDK) that allows messaging between different devices through publish/subscribe mechanism. As of now, it is divided into 2 different generations, ROS1 and ROS2, with both having numerous versions, or distributions (distros) as they are called. The latest distros are Noetic for ROS1 and Galactic for ROS2 at the time of writing. [11–13]

Initially ROS began in 2007 in Stanford University as a bundle of several robotic software frameworks in research projects. It has since grown into a massive community with tens of thousands of users ranging from hobbyists to professional industrial automation companies [14]. Market estimates also expect this number to only rise in growth in the future [15, 16]. ROS development is also supported by some large and well-known companies in the industry, such as Amazon, Intel, Bosch, Microsoft, LG Electronics and Toyota Research Institute among others [12].

Although ROS1 and ROS2 have many fundamental differences, the core concepts, described in the Section 2.1, are the same in both of them. The most notable differences are covered in the Section 2.2.

2.1 Core concepts

Any device that uses ROS to communicate in a ROS network is called a node. Nodes can send and receive messages through named topics. Besides through topics, nodes can also interact through services, actions and parameters. According to ROS philosophy, a node “should be responsible for a single, module purpose”, meaning a single robot would commonly have multiple nodes in it. One node in a robot could be responsible for the wheel motors, another node for the camera and so on. The following subsections describe the other core concepts and functionalities of ROS systems. These concepts apply to both ROS1 and ROS2. [13, 14]

2.1.1 Topics

When a node sends messages to other nodes, in ROS terms, it publishes a message to a topic. All the nodes that want to receive messages published to a topic subscribe to that topic. A single publish into a topic sends the message to all nodes that have subscribed to the topic. As such, the messaging through a topic can be point-to-point, many-

to-one, one-to-many or many-to-many in nature. The messaging functionality is depicted in the Figure 1. [13]

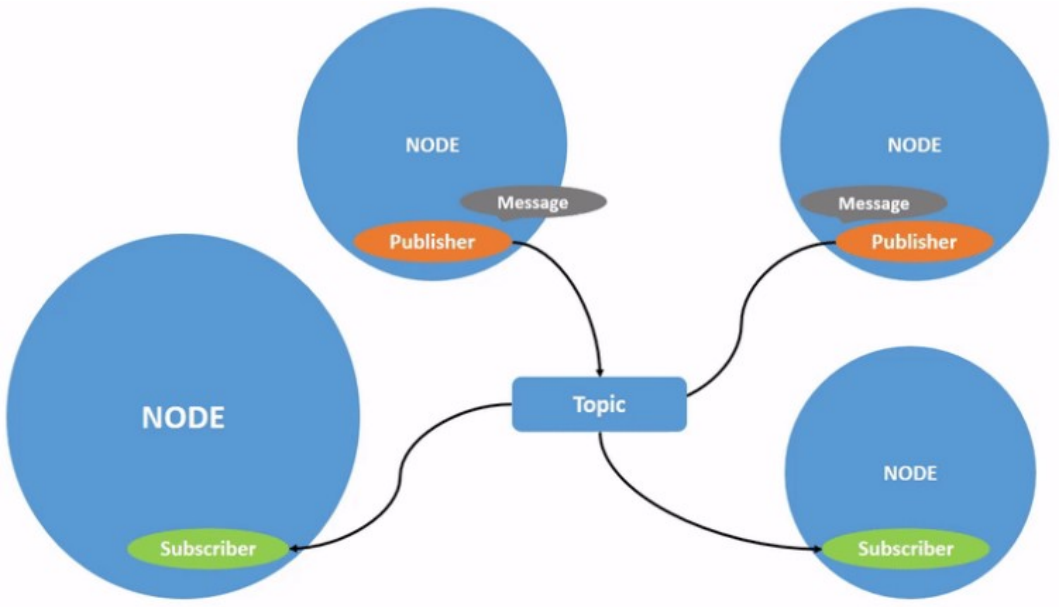

Figure 1. *ROS messaging through topics* [13]

2.1.2 Services

Unlike topics in the publish-subscribe model, the services are based on the call-response model. In other words, a service is called on a node and the calling node receives a single response from the called node. Multiple nodes can call the same service but there can be only one service server for a given service (with a given service name). ROS service functionality is depicted in the Figure 2. [13]

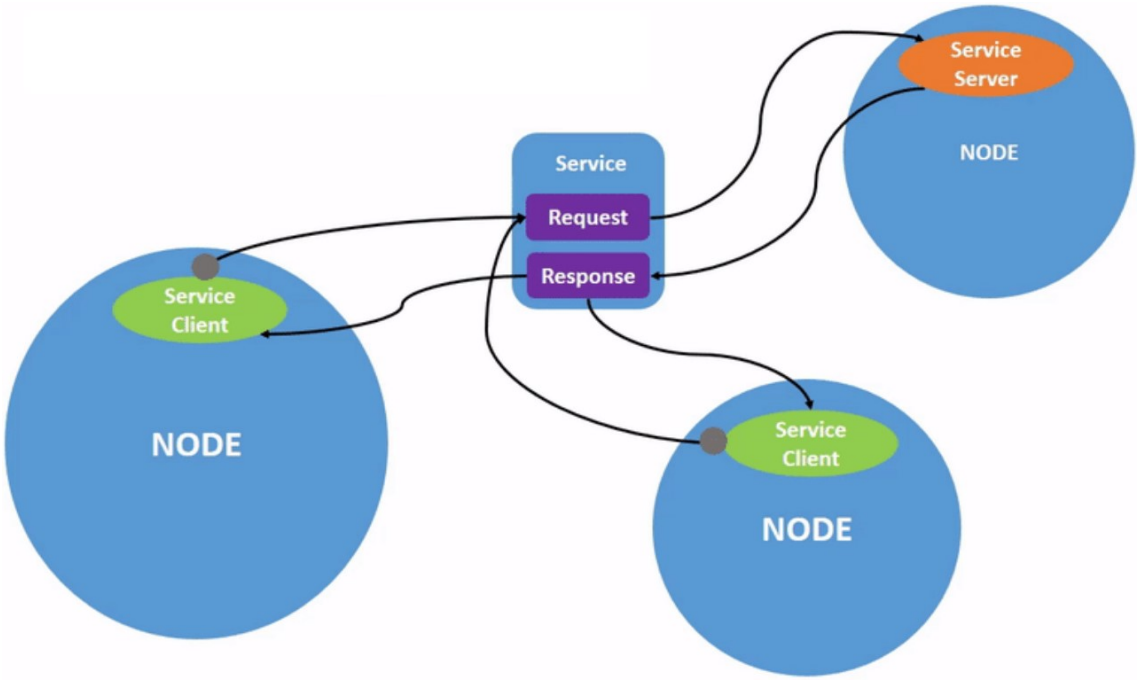

Figure 2. *ROS communication through services* [13]

2.1.3 Actions

Actions utilize topics and services and are intended for long running tasks. They consist of 3 steps: goal, feedback and result. First, a goal is sent to the action server through a goal service, and the action server acknowledges it by sending a response. Next, a result is requested from the action server through the result service, which makes the server start the action procedure. During the action procedure the action server gives a steady feedback of information through a feedback topic. Unlike services, actions can be cancelled during the procedure. When the action is finished or interrupted, the action server sends a result through the result service. Action functionality is depicted in the Figure 3. [13]

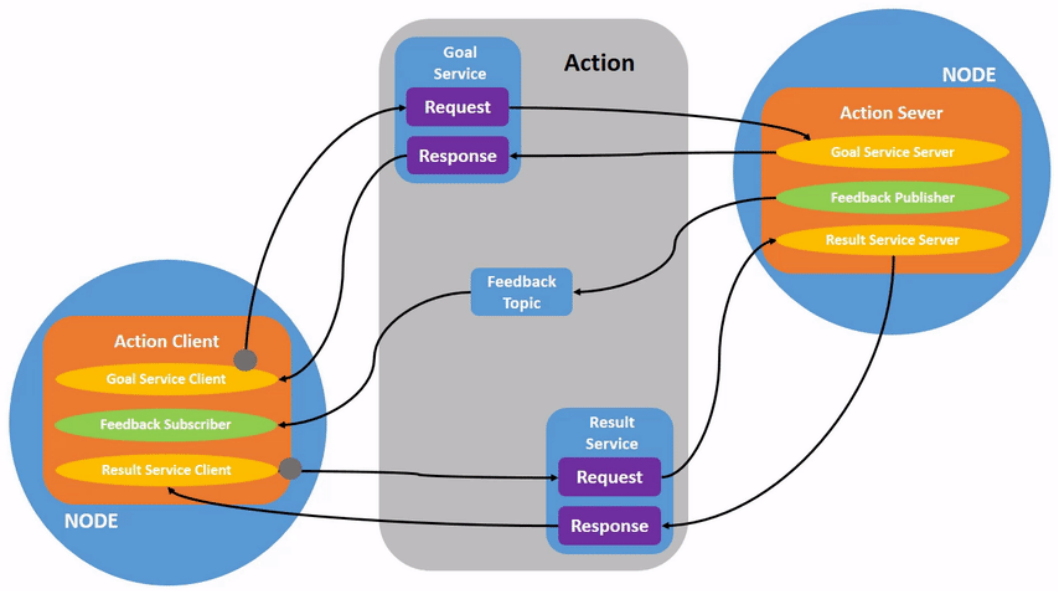

Figure 3. *ROS messaging through actions* [13]

2.1.4 Parameters

All nodes in ROS have configuration parameters that can be accessed and changed at runtime. The parameters can be accessed and changed through command line tools, launch files or through software. The parameters can also be saved or “dumped” into a file to be able to save specific configurations for inspection or later use. [13, 17]

2.1.5 ROS graph

Any ROS system can be depicted by a ROS graph, which shows all the nodes and their relations with each other through publishes and subscriptions topics. In the Figure 4 is an example of a ROS graph. In the graph, ovals are nodes, rectangles are topics. There might also be rectangles that encase nodes and topics in a ROS graph. Those would depict namespaces. Arrows in the graph go from publisher(s), to topic, to the subscriber(s) [18]. The nodes in the graph are from a Turtlebot3 Waffle Pi and a laptop. The

node /_ros2cli_2942 is a laptop subscribed to /imu topic through command line tools. Other nodes are from the Turtlebot3. [13, 14, 18]

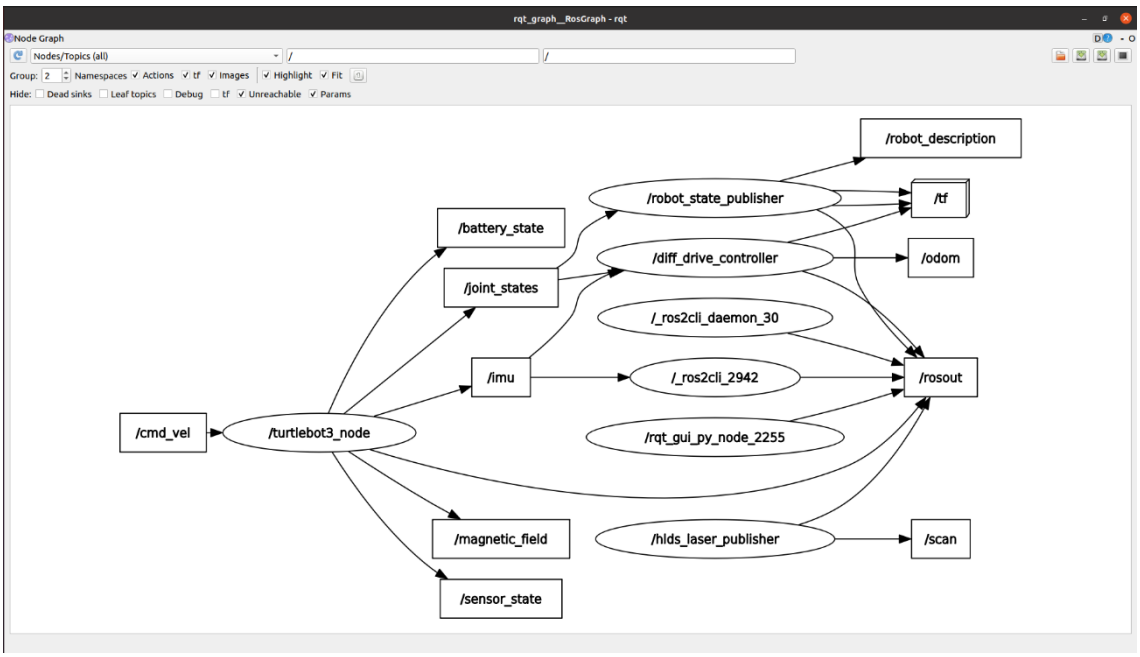

Figure 4. *ROS graph example.*

A few topics of interest in the graph above are /cmd_vel, /odom and /rosout. /cmd_vel (command velocity) is a topic that allows publishing linear and rotational velocities for the robot, which make the robot move accordingly. /odom (odometry) can be subscribed to receive the position and orientation of the robot in relation to its idea of a world origo (the (0, 0, 0) point in the world), which is commonly the position the robot was at when it was turned on. /rosout is a topic to subscribe for logging and debugging messages. This feature can be utilized in the C++ and Python APIs, when creating custom nodes in both ROS1 and ROS2. With Turtlebot3 however, there seems to be no messages to be received from /rosout by default, even with all the nodes having a publisher to it. [13, 14, 19]

ROS is mostly used for robots, but technically it can be used for any kind of sensor or actuator network with or without robots as it has software packages that allow support for various kinds of sensors, like a Light Detection And Ranging (LIDAR) or depth camera for example. Some software algorithm packages are especially made for robots however, such as Navigation and Simultaneous Localization And Mapping (SLAM). SLAM allows a robot to simultaneously map its environment and position itself in it. In other words, with SLAM it is possible to make a map of an environment by driving the robot in the environment, while also getting the position of the robot in the environment. Note that

SLAM is the name of the algorithm, but implementations of the algorithm may vary. Navigation utilizes SLAM to conduct navigation in the mapped environment using various different navigation algorithms. [20–23]

ROS also includes an array of utility software for visualizing the ROS network or running simulated ROS environments. Most notable software for ROS are Gazebo, TurtleSim, RViz and RQt. Gazebo allows simulating environments populated with robots. It includes an OGRE graphics engine, multiple different physics engine options, multiple robot models and multiple plugins and tools. The aim is to be able to do robotics development with just software based simulation. An example Gazebo scene is seen in Figure 5. [13, 24]

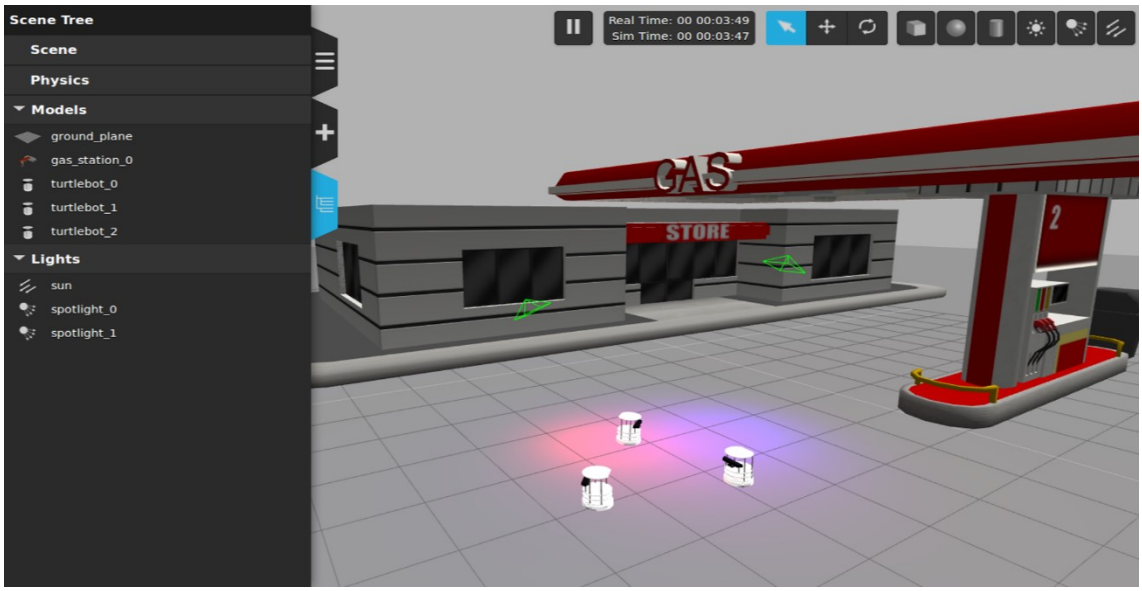

Figure 5. *An example Gazebo scene* [24]

TurtleSim is another simulation tool, but where Gazebo is for serious development, TurtleSim is just a lightweight simulator meant for learning ROS. As the ROS2 documentation puts it: “It [TurtleSim] illustrates what ROS 2 does at the most basic level, to give you an idea of what you will do with a real robot or robot simulation later on” [13]. In TurtleSim, there is a 2D scene with a turtle moving in a 2D plane. The turtle in the scene is subscribed to a ROS topic that allows it to move in the scene. This can be done through the command line tools, RQt, a custom package or a teleoperation software in the TurtleSim package. In Figure 6 is an example of TurtleSim scene. [13, 18]

Figure 6. *An example TurtleSim scene*

RViz (ROS visualization) is a 3D visualization tool, which allows visualization of what a robot is sensing, “thinking” and doing. It understands state and sensor information, such as camera, point cloud, laser scan and coordinate frame data. The different sensor information have specific displays that allow the user to choose how to visualize the data. RViz allows the programmer to also set visualization markers, such as custom colored cubes, lines and arrows, at will. The idea behind RViz is to visualize the numerical data in a 3D environment that would otherwise be difficult to comprehend, design and debug. RViz has been ported from ROS1 to ROS2 with the name RViz2 and most of the features of the original available. RViz2 is part of the official ROS2 installation package. Figure 7 shows an example scene from RViz, where a robot has the camera data on upper right corner and laser scan data in the large picture. [14, 25]

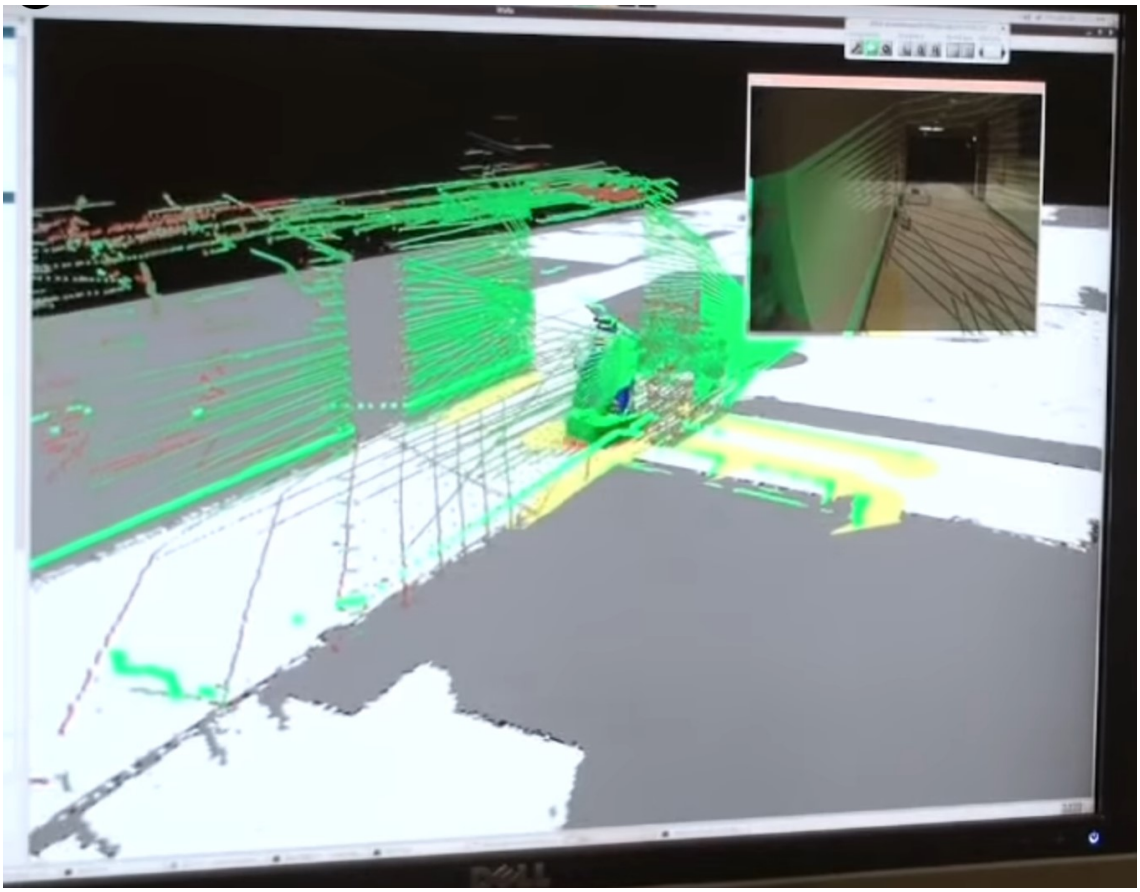

Figure 7. *An example RViz scene* [14]

RQt is a plugin based graphical user interface tool for ROS2 and is included in the ROS2 installation package. The plugins that are included in the ROS1 version of RQt are listed in ROS1 Wiki [26]. Arguably the most notable plugins in RQt are the node graph, topic monitor, message publisher, plot and image viewer, which are all also ported to ROS2. The node graph shows the node graph of ROS network, as explained and depicted in the Subsection 2.1.5. Topic monitor allows for graphical inspection of messages being published and message publisher allows publishing messages to any topic. Plot allows for plotting any message values into an X-Y graph. Image viewer, as the name implies, allows for subscribing to a topic of any image message type to show the images being published. In the Figure 8 a simple RQt scene can be seen, which shows topics of a TurtleSim node. The plugins shown in the figure are topic monitor, service caller and message publisher. [13, 18]

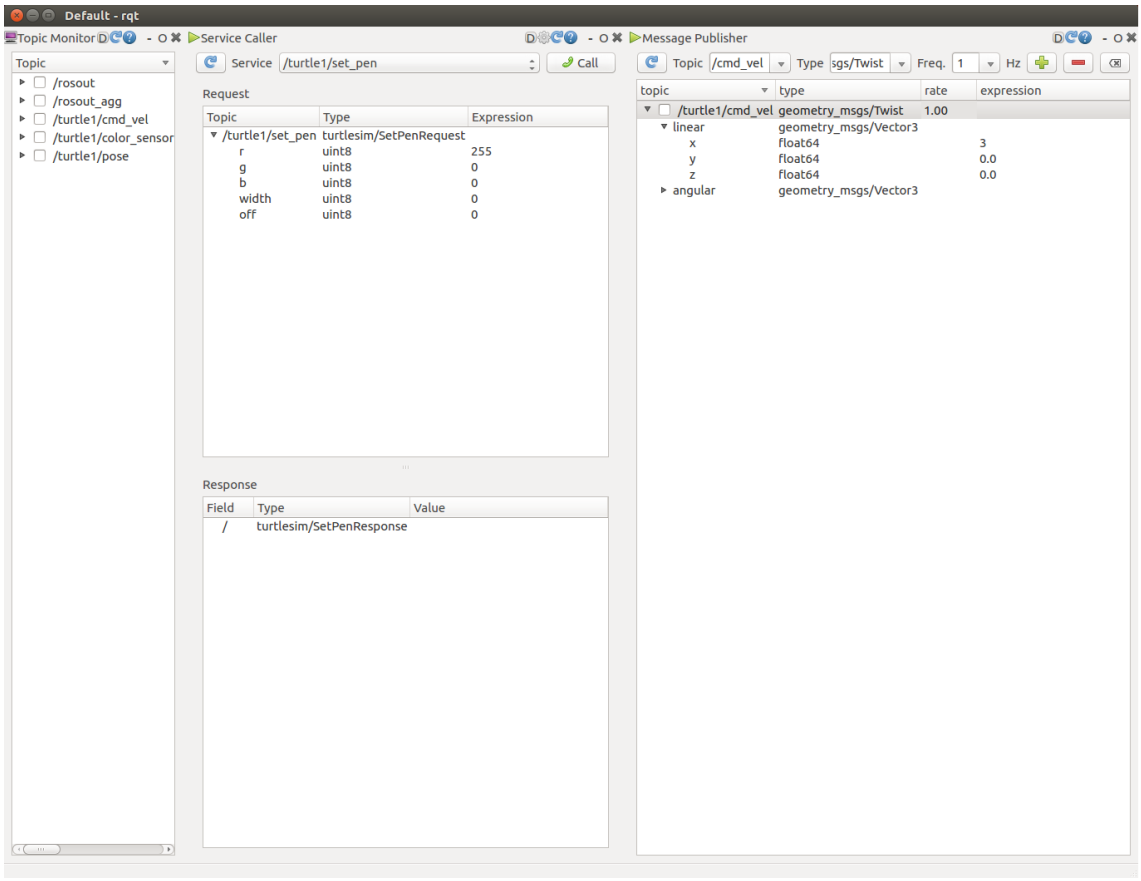

Figure 8. *An example RQt scene with TurtleSim*

2.2 Differences between ROS1 and ROS2

The core differences between ROS1 and ROS2 are covered in this section and listed in the Table 1. The list is not exclusive however, as there are too many small changes to cover in this thesis.

Table 1. *Core differences between ROS1 and ROS2*

Features	ROS1	ROS2
Network security	No security	Built-in
Middleware	Custom middleware	DDS
Network model	Through a broker (ROS master)	Distributed
Quality-of-service	No	Yes
Real-time support	No	Yes
Official platform support	Ubuntu	Ubuntu, OSX, Windows 10
Programming language	C++03, Python 2 (Python 3)	C++11, Python 3.5 =<

One of the most pressing issues with ROS1 is that it doesn't have any kind of security measures implemented into it. This is one core change in ROS2 that was especially targeted for industrial applications. ROS2 utilizes data distributed service (DDS) middleware, which was already an established open source standard protocol before the adoption to ROS2 and also includes many security features. DDS is maintained by Object

Management Group. As it is an open standard, there are multiple vendors for DDS software, both commercial and open source. ROS2 supports a commercial DDS software RTI Connext and 2 open source DDS software: Eclipse Cyclone DDS and eProsima Fast DDS. Up to the distro Foxy, Fast DDS was the default DDS software, but in the most recent distro, Galactic, Cyclone DDS became the default. [11–13, 27–31]

A ROS1 network always has a ROS master node. This node acts as a broker, registering node, topic, service, parameter and action names. Without it, other nodes could not find each other. This is considered troublesome for the robustness of the system as the ROS master acts as a single point of failure. If the ROS master crashes, no new connections between nodes can be made, unless special measures like distributed multithreaded checkpointing (see [32]) are taken. These kind of special measures are outside the scope of this thesis however. [14, 32]

In ROS2 the DDS middleware allows dynamic discovery of nodes in the network and dynamic addition of publishers and subscribers into the network. With DDS networks there is no single point of failure, like in ROS1. [13, 27, 28]

Another feature of DDS that is new to ROS is quality-of-service (QoS). QoS has been formalized as a native part of ROS2, so users can configure the QoS parameters in ROS2 instead of digging into the underlying DDS options. Figure 9 shows how QoS functions in ROS2. Publishers offer certain QoS and subscribers request specific QoS. If these are compatible, a subscription is created, otherwise no subscription is created. [13, 33]

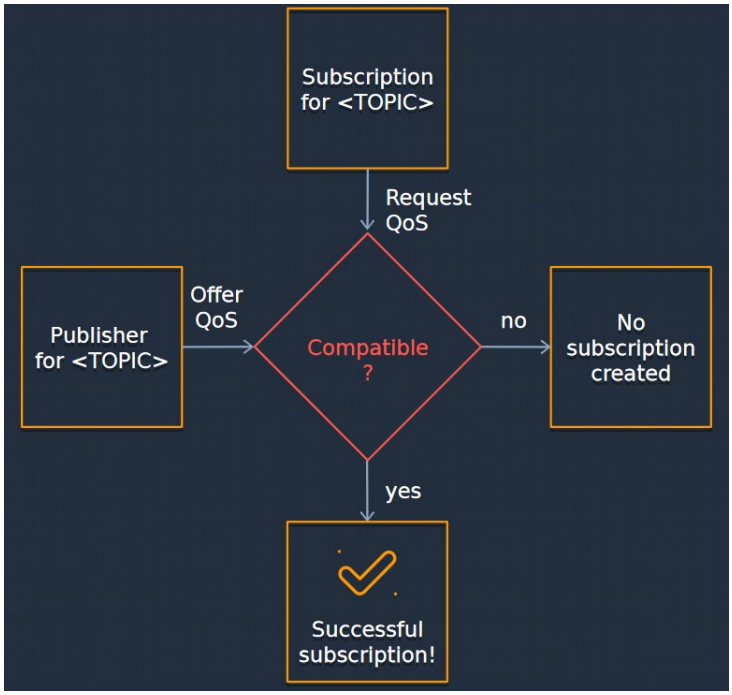

Figure 9. *QoS functionality* [33]

QoS policies include history, durability, reliability, lifespan, deadline and liveliness. For publisher and subscription to be compatible, all the policies chosen must be compatible. History decrees how many messages are kept locally. This accounts to both subscribers and publishers. Durability decrees if the publishers should provide old messages to a fresh subscription. Reliability decrees if messages have to be delivered/received reliably, or if best effort is enough. Basically this comes down to the choice of using TCP or UDP as the network transport layer. Lifespan decrees how long should a publisher keep an unsent message before it is discarded as being not useful. Deadline and liveliness utilize a new feature called QoS event callback. This is essentially an interrupt function for when an event happens with QoS. Deadline decrees how often (at which frequency) should a publisher send messages. If a deadline is missed, a callback event is called for both the publisher and the subscribers. Liveliness decrees what kind of a heartbeat should a node or a topic give to prove it is still working (alive). If a heartbeat deadline is missed a callback event is called on both the publisher and the subscribers. [13, 33]

Another requirement that the robotics industry often requires is real-time communication. This was not thought of in the design of the initial ROS1 and was not added into it later to not break the API that many members of the ROS community already relied upon. ROS2 however had no APIs to break when it was designed and introduced, so it allowed the consideration of real-time communication in the redesign process. ROS2 documentation includes a brief tutorial on real-time programming on ROS2 using Xenomai or RT_PREEMPT with Linux. [13, 34]

The build system and platform support are also broadened, as ROS1 was only tested on Ubuntu Linux platform and only supported CMake build system. ROS2 is tested on Ubuntu, OSX and Windows 10 and officially supports Python packages besides CMake builds. ROS1 with Python supported Python 2, but as Python 2 has reached is end-of-life (EOL) in 2020, ROS2 was built to only support Python 3.5 or higher [35]. The last ROS1 distro, Noetic, also supports only Python 3 [14]. The C++ utilization is also upgraded as ROS1 utilizes only C++03, while ROS2 uses C++11 and some parts from C++14 in its API. [13, 36]

2.3 ROS1 to ROS2 transition

The transition from ROS1 to ROS2 was driven by new features up until May of 2020, when Open Robotics declared their last ROS1 distro, Noetic. This is a long term support (LTS) release, so it is supported with updates until May 2025, but after that, no new

ROS1 distros are coming and Open Robotics is concentrating all its development towards ROS2 from that point onwards. This should effectively drive the ROS1 community towards ROS2 in the coming years. [14, 37]

There is still room for improvement in ROS2 as reportedly in 2020 there were still many bugs around, tools were not always working as expected and the documentation was sparse [10]. Although as ROS2 development has gone a long way in the past years, many of the bugs and tools could be fixed already or in the near future [38, 39].

The following subsections cover the different aspects of the ROS1 to ROS2 transition.

2.3.1 Studies and recommendations

Some entities are suggesting to start making the new projects in ROS2, while the porting of older projects to ROS2 might still be debatable in some cases [12, 40, 41]. New projects should be done in ROS2 as this will reduce future technical debt. Also as there have already been 2 LTS ROS2 distros, dashing and Foxy, there should not be any new features coming rapidly that could break the API functionality. As the last ROS1 distro Noetic reaches its EOL in 2025, only temporary (projects not used after 2025), hobbyist or academic projects in ROS1 are not considered to be in a pressure to migrate into ROS2. Even so, academies should start considering a ROS2 based curriculum for the future and even hobbyists should start slowly checking out the ROS2 ecosystem. [12]

There doesn't seem to be any study or report that would state that ROS2 is now ready for hard adoption by the industry yet. Many reports have stated that ROS2 is in the stages between "maybe" and "depends on what you are doing". These reports are from around 2019 to early 2020 however, as there seems to be no evaluation on the state of ROS2 as a whole after that. Considering how ROS2 was around 2019 with the Dashing and Eloquent distros however, there have been rapid improvements. The problems these studies report include performance issues with the DDS layer, unavailability in features and bugs in essential tools like RViz and RQt, missing drivers, unmigrated software packages and sparse documentation and tutorials. [10, 42–44]

As for the DDS layer performance problems, there have been some studies about the performance of different DDS implementations recently. There have been noticeable differences in latency for the different DDS-middleware available for ROS2, while the type of hardware also played a part [40, 45]. One of the inducers of problems with DDS was reportedly the QoS parameters [10]. If a publisher would have different QoS parameters than the subscriber, the message would be discarded without any notice. With Foxy distro, this was fixed to give an error if QoS parameter conflict happens [46].

As for RQt, there were reportedly plugins that were not migrated from ROS1 yet in Eloquent. Looking at RQt in Galactic, this seems to still be the case, although many of the essential plugins are already migrated. RViz reportedly has most of the essential features migrated however. In reports later than Foxy distro, there has been no complaints of bugs in RViz or RQt, which makes a case for them being patched. This is not conclusive however, as they were not extensively tested as part of this thesis. [42]

The problem with missing drivers is slowly starting to be fixed as manufacturers have started porting their drivers to ROS2 and a few have already released a beta or release version of their ROS2 drivers. More on this in the following subsection.

Some reports were raising the issue of important ROS packages not yet being migrated to ROS2. At the time many of the popular packages needed to be built from source [42]. Many of these are now available as debian packages for ROS2 under Ubuntu. Some packages, like MoveIt, were also in beta stage at the time of the reports, but have now come under official release.

The documentation is still rather sparse with ROS2, with most of the tutorials covering simple messaging examples, but not how to expand on that. There is also a separate documentation for the C++ and Python APIs, but during their use on this thesis, this was often too sparse and sometimes out of date and the function interfaces needed to be checked directly from the source code [17, 47]. Nevertheless, there are also complaints on the documentation being of a level that expects expert programming background [10]. This is definitely a part of ROS2 that requires improvement in the future, to simplify and expand the documentation.

2.3.2 Industry adaptation of ROS2

ROS1 development was overseen by only one organization effectively, initially Willow Garage and later Open Robotics, which was created by Willow Garage [48, 49]. With ROS2 however, Open Robotics has decided to form ROS2 Technical Steering Committee, to set the direction ROS2 development is going. This organization had founding members from Microsoft, TARDEK, Amazon, Intel, LG Electronics, Bosch, Apex.AI, ARM, Toyota Research Institute, Open Robotics and ROBOTIS. With powerful companies in the industry investing in ROS2, the expectations for the future of ROS are high. [50]

In late 2020, there were already multiple robot manufacturers supporting ROS2, including Turtlebot3 by Robotis, ROSbot by Husarion, Rover Zero by Rover Robotics, Hadabot by Hadabot, e-puck by GCTronic and Open Manipulator by robotis. In April 2021, the first collaborative arm robot (cobot) ROS2 open source driver package in the industry

was released by Doosan Robotics [51]. Right after Doosan, Universal Robots (UR), PickNik, and Forschungszentrum Informatik released the beta version ROS2 drivers for UR's cobot arms in May 2021. This is major news as UR is the leading developer of cobot arms owning roughly 40% of the entire market. [52]

There was also a company called Acutronic Robotics released the MARA robot that utilized ROS2 already in 2018. The core product of Acutronic Robotics was the H-ROS modules however and MARA was only sold as an example of what H-ROS can do. The idea of H-ROS would be that all robotic modules, such as arms, wheels, sensors etc. would have this hardware chip which would communicate with other modules and the user/controller through ROS2. This would allow completely modular design, as hardware from any manufacturer would work as part of a robot seamlessly as the communication would be through a common ROS2 network, as long as there would be an H-ROS module available for that hardware. This would also be controllable by any ROS2 enabled computer. Acutronic Robotics shut down however on July 31 2019, because of lack of financing. The CEO of Acutronic Robotics Victor Mayoral speculated that they "hit the market too early and fell short of resources". [53–57]

NVIDIA has released a robotics simulator environment Isaac Sim in June of 2021, which supports ROS2. The software is built on top of NVIDIAs Omniverse simulation platform. The idea is rather similar to that of Gazebo allowing robot simulation with custom environments. Isaac Sim advertises photorealistic graphics with real-time path and ray tracing, domain randomization, multi-camera support, multiple robot import formats, GPU-enabled physics simulation, material definition language support and variety of sensor type support. [58, 59]

ADLINK in cooperation with Intel and NVIDIA have released a Series of controllers that support ROS2. The features vary from low energy consumption with high AI computing capabilities to x86-64 architecture CPU with a wide array of I/O ports and integrated real-time mechanisms. The prices of these platforms also vary from around 2100 to 5400 $, depending on the features [60, 61]. This effectively signals that this series is for industrial use rather than hobbyists. [62, 63]

2.3.3 ROS1_bridge

ROS1_bridge is a packet for ROS2 that translates messages from topics and services across ROS1 and ROS2 both ways. It has the problem of being a single point of failure in a system stretched across ROS1 and ROS2. In addition, it can have a high CPU load because of message translation and also induces latency. ROS1_bridge supports the built-in message and service types by default. If custom message types are required,

ROS1_bridge needs to be built from source. This requires naming the custom message packets in a way they can automatically be translated. If this is not done, or the message type fields are not the same in ROS1 and ROS2, there is also possibility of making custom translation of messages by defining it in mapping rules YAML file. ROS1_bridge was briefly tried and tested as part of this thesis. This is covered in more detail in Sections 5.3 and 6.4. [42, 64, 65]

Ideally, ROS1_bridge allows companies to migrate their software one module at a time assuming the same message types are used, as the migrated modules can communicate with the non-migrated ones through ROS1_bridge. If this approach is taken, the latency issues covered in 6.4 should be taken into account. [65, 66]

2.4 Software interfaces

This section covers only the official ROS2 software interfaces and their core functions. There is much more functionality with the various ROS2 software packages available, but that is beyond the scope of this thesis.

ROS2 has 3 major programming language APIs: Python, C++ and C, although only Python and C++ are mentioned in the documentation and covered in tutorials. The APIs are built on top of ROS client support library (RCL) package and named accordingly as *rclpy* for python, *rclcpp* for C++ and *rclc* for C. RCL has some common core functionalities for ROS2 and it is designed to “support implementation of language specific ROS Client Libraries”. [13, 67]

rclc complements RCL, as RCL is also written in C. Unlike *rclcpp* and *rclpy*, *rclc* doesn’t add a layer of types on top of RCL, but provides functions to make programming with RCL types easier. Also unlike *rclcpp* and *rclpy*, there seems to be no separate API documentation page for *rclc*. [67]

rclcpp wraps the ROS2 node into a class, where custom nodes can be inherited from. This is the way presented in the ROS2 documentation tutorials. The node class has methods for creating publishers, subscribers, service clients and service servers. Actions also have their own functions for creation of servers and clients, but they are wrapped in another package, *rclcpp_actions* [68]. There are also methods to make timers and to create and modify the node parameters. For callbacks there are also many different components and functions. [13, 17, 67]

rclpy also wraps the ROS2 node into a class, where custom nodes can be inherited from. This is also the way presented in the ROS2 documentation tutorials. Like *rclcpp* the *rclpy* node has similar methods for publishers, subscribers and service clients and servers,

timers, parameters and callbacks. In *rclpy*, the classes for creating action servers and action clients are in the same library however. [13, 47, 67]

There are also ROS2 APIs for multiple other languages such as Ada, Java, NodeJS, Go, Objective C, .NET and Rust [69–75]. These are not official parts of the ROS2 yet however, but rather individual projects by the ROS community contributors. As such, there might be errors with them, the documentation is sparse and there is no guarantee of continued support for them, even though there is no reason to believe these projects would be shut down either any time soon.

2.5 Build tool

This section covers the ROS2 build tool and its core aspects. As a distinction, in this thesis a build system is considered to configure, build and install a single package, whereas build tools are considered to utilize build systems with added functionality, such as building multiple packages in a topological order.

In ROS1, the original build tool was *rosbuild*, which got superseded by *catkin_make, catkin_make_isolated* and *catkin_tools*. The tools provided some additional functionality on top of the CMake build system. The custom build tools were deemed necessary because ROS projects are often built from numerous different packages, which have dependencies on each other, and micromanaging the build order and parameters on these packages would have been very difficult and tedious.

For ROS2, *ament_tools* was the initial build tool used up to Ardent distro, but since then *colcon* has been the build tool recommended. The core benefit of *colcon*, compared to its predecessors, is the support for multiple different build systems besides CMake, which, together with RCL, allowed adoption of other programming languages and packages that were not made for CMake. [76, 77]

The design goals of *colcon* are to make building, testing and using multiple packages easy, to enable any kind of build system support through extensions, allowing any kind or package to be built without altering the sources and to make built packages immediately usable. There are a few features specifically stated to be outside the scope of *colcon* however. These include fetching the sources of packages, installing dependencies of packages and creating of package level binary packages. Reason stated is that tools for those tasks already exist. [76, 77]

In ROS2 documentation the build procedure suggested is to check possible missing dependencies with *rosdep* and then invoke the *colcon build* command, which builds the package. *rosdep* is a separate tool for checking package dependencies and it only works

on Linux. By default *colcon build* will only show the status of the build when running and list the return value and build time of packages built. If any package should fail to build, *colcon* will abort the rest of the packages and exit without giving any additional information about the error. The output log of the build can be checked from log folder in the package root folder or it can be received directly to the console window by invoking *colcon build* with a specific parameter. [13, 76, 77]

3. EXTENDED REALITY

Extended reality, sometimes also referred to as cross reality, is an umbrella term, including virtual reality (VR), augmented reality (AR) and mixed reality (MR). VR is a completely computer generated environment with no real world aspects included while AR and MR are mixes of real world and virtual aspects. The difference between AR and MR is the immersivity. In AR the virtual aspects do not need to necessarily blend in with the real world or be interactable, while in MR the goal is to completely blend the virtual aspects into the real world. This is illustrated in Figure 10. [78, 79]

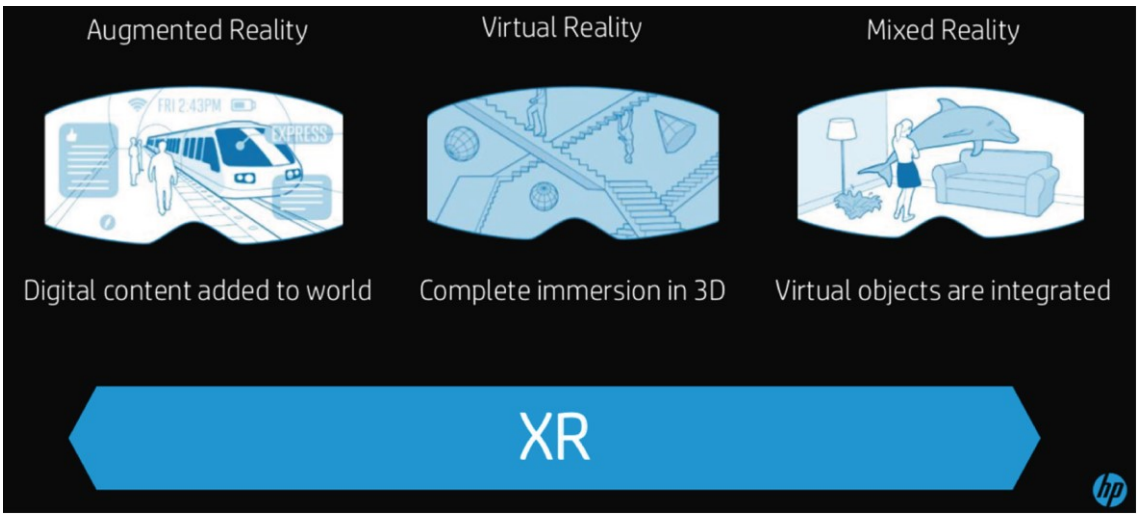

Figure 10. *XR, VR, AR and MR* [78]

These are still rather ambiguous concepts on a scale from the perceived real world reality to completely immersive virtual reality as seen in the Figure 11.

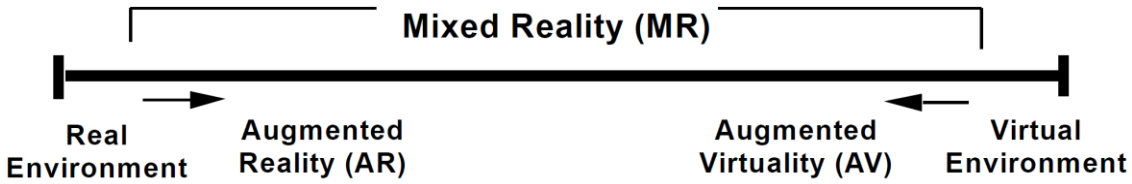

Figure 11. *The Reality-Virtuality continuum* [80]

Note that in Figure 11, the name Mixed Reality was suggested as the umbrella term for AR and VR in 1994, but as of 2021, the name Extended Reality has been adopted instead [79, 80]. As opposed to AR, Figure 11 also introduces the term Augmented Virtuality, meaning a virtual environment augmented with real world aspects. It has not become a common term however in the general discussion of XR. While the XR technology can be made with any kind of sensory input that immerses the user, most product innovations in 2021 are done on visual or audio-visual virtuality. In 2021 the head mounted

display (HMD) seems to be the most common type of interface in XR, allowing audio-visual immersion. Other interface types are often composed of traditional computer displays in various forms as seen in the Section 3.2.

XR systems aim to serve multiple different customer segments, including simulation, surveillance, planning, design, gaming, entertainment in general, training and maintenance [81]. The reasons to why XR is so lucrative are many. With just the human need of experiencing different scenes and worlds with high immersion, lies a huge market for entertainment and games. However, in 2019 the industry had already overtaken the game industry on XR spending according to IDC ([82] as cited by [83]). The possibilities of XR in the industry are vast. In many different fields, training can be done much more effectively, safely and sometimes even cheaper with XR than traditional methods. Simulation of events and design objects as interactable virtual 3D objects is much more descriptive than a traditional 2D model on a display. Also material consumption in many areas, such as scenery/interior architecture or any other design heavy area, are diminished, as design iteration can be made virtually without needing physical prototyping. Less material consumption might also lead to increased sustainability. Furthermore, safety is also improved with XR, as many safety critical procedures can be rehearsed with XR before the real thing. [84, 85]

A major problem with XR technology, which has persisted, is simulator sickness, also known as cybersickness. These are terms used when users of XR devices feel eye strain, nausea or disorientation during or after the use. The causes of cybersickness are not known exclusively, but there are multiple suggested causes. The most widely cited theory is sensory conflict, where a mismatch between the signals that the visual, auditory, tactile, kinesthetic, and vestibular sensory systems give induces discomfort. There are multiple ways to reduce cybersickness, such as better displays and more natural locomotion or lack of locomotion in virtual environment. One popular design trick, used to move around in VR without actual locomotion, is teleporting. Display features are described in the Section 3.1, while some locomotion innovations are covered in the Section 3.3. [86–89]

As of now, XR as a technology is still in an evolving state with displays, sensors and networking technologies being refined year-by-year. Especially 5G technology is expected to heavily improve XR prospects in coming years [83, 90]. The trend of XR is however on the rise as the global market value of XR is expected to multiply in the coming years [91–93]. This can also be seen in the news as many powerful companies like Qualcomm, are investing in the evolution of XR [94].

3.1 Head mounted displays

On one hand, HMDs may be either see-through or closed systems and on the other hand they may be either tethered or standalone. In see-through systems the users can see the real-world through glass-like goggles, as opposed to closed systems which do not use see-through material for visualization. Tethered displays are connected to a separate PC or another computer device with a cable, whereas untethered systems are standalone without cables or outside computing. The tethered system's PC's also have rather high computing requirements [95]. In the Figure 12 are examples of see-through (Microsoft hololens2) and closed (HTC VIVE Pro) HMDs.

Figure 12. *Example HMDs: Microsoft Hololens2 (left) and HTC VIVE Pro (right)* [96, 97]

The interaction methods with HMD's are also varied, with some systems allowing hand or eye tracking, while other systems operate with specific, often hand-held, controllers. The capabilities of the wide range of devices and accessories are usually very specific for the use case that they are designed for, like gaming. Some high-end devices aim to be more general in their use case however and cater to a wide array of use cases, like Varjo devices and Microsoft Hololens 2 for example. There are also multiple ways for the hardware to interact with the user. This is called feedback and it includes visual cues, sound feedback and various kinds of haptic feedback. Some high-end devices even allow spatial mapping, which allows 3D digitization and registration of the real-world into a virtual 3D model, opening a variety of new ways to implement augmented and mixed reality.

See-through systems are suitable for AR/MR solutions by desing, as the user can see the real world through the lenses in addition to the added virtual content. Some modern closed displays can also work as AR/MR devices if they have external cameras allowing the user to "see through" the display. Closed displays allow also full VR solutions.

The benefit of see-through HMDs is that the user can commonly see the real world through the glasses in higher resolution than is achievable with cameras and displays, while still including virtual elements. Because of the nature of see-through HMDs as of now, the rendered objects are all transparent, like holograms, while with closed headsets, one can see also solid looking objects.

Closed HMD's in AR have a rather high computational need, as they need to have extensive camera stream with low latency from the outside of the headset, while adding virtual graphical objects on top of that. The advantage is that, compared to see-through solutions, the virtual objects can look solid, which adds to the immersion. Closed HMD's with cameras thus allow a free spectrum of applications from the reality through a camera to a complete virtual reality.

3.1.1 Refresh rate and resolution

The displays in headsets have a few important metrics to consider. A common refresh rate for VR displays is around 90 Hz, while it does range from 70 to even 180 Hz. The refresh rate is perceived by the users as how smooth is the motion of objects is in the display. Refresh rate is more important in XR than in regular displays, as lower refresh rates of 75 Hz or below are reported to induce simulator sickness [98]. [87]

The resolution of the screen is another major factor in displays, as it is also associated with simulation sickness. Studies show that a resolution lower than 960 x 1080 pixels per eye are likely to induce simulator sickness [98, 99]. The effects between high and very high resolutions are not mentioned in this study however, as very high resolution HMDs are a rather recent phenomenon. The modern HMD resolutions vary from around fullHD resolutions (1920 x 1080) to UltraHD resolutions (3840 x 2160) per eye. The aspect ratios vary too from the traditional display ratio of 16:9 to 1:1 ratio. A high resolution also creates a clearer picture, which increases immersion. However, there are also other metrics that account to the picture quality, such as subpixel layout, colour reproduction and lens optics [100]. [101]

The resolution is closely tied to the field of view (FOV), which is the angle of view in the display around the eyes. This can be measured separately in horizontal and vertical axis or diagonally. An accurate value of FOV is hard to measure in a headset, as the fit and facial features are varied in every individual and they affect the perceived FOV. Still the FOV values that manufacturers publish give some indication on the matter, as there are HMDs with claimed FOV values ranging from 90° to 200°. Studies show that a FOV of less than 110° might induce simulator sickness in VR [98]. A larger FOV is not all good however, as there is a set resolution for the screen and a larger FOV spreads that across

the display, which makes the picture look more granular. Therefore another metric of angular pixel resolution is used to accommodate the connection between resolution and FOV. This measures the amount of pixels per degree and is therefore a more accurate measurement unit for the quality of the picture in XR. [100, 101]

3.1.2 Color

Besides the different forms, resolution and smoothness of the displays, the colour quality also plays a pivotal role in immersion. The displays used in HMD's are in this sense the same as traditional monitors. The metrics to pay attention with colour are colour gamut, colour resolution and display panel type. Colour gamut tells how large area of the human discernible colours can be produced by the display, as seen in Figure 13. There are a few colour gamut standards to help comparing displays, like sRGB and Adobe RGB. These are usually used as reference with display specifications, such as "covers 98% sRGB", means the display covers most of the sRGB standard area of colours. The colour resolution describes in essence how many different shades of colours are available, higher resolution resulting in more unique colour shades. In other words, it tells into how many different usable shades is the whole color gamut divided into. Common colour resolutions are 6-, 8- and 10-bit, meaning the amount of bits used to encode each color channel. For example, 8-bit color encoding uses 8 bits for red, green and blue channels, totaling 24 bits per pixel.

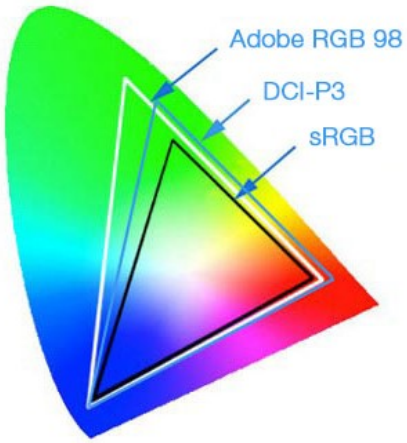

Figure 13. *Common colour gamut standards* [102]

The display panel types used for HMD's are commonly divided into LCD and OLED panels. Furthermore, LCD panels are divided into TM-, IPS- and VA-panels. Roughly generalizing, TM-panels are very fast in response time and cheaper, but have poor colours, VA-panels have slow response times, good contrast ratio and mid-range colour quality, IPS-panels have mid-range response times, mid-range contrast ratios and good colours. OLED panels on the other hand have great contrast and great colour but increased cost,

shorter lifespan and a risk of screen burn-in. With VR headsets OLED displays are commonly made with the PenTile matrix pixel layout, which has less sharp picture quality compared to same resolution LCD or OLED screens with the RGB pixel layout as a PenTile display has 33% less subpixels compared to an RGB display. A PenTile OLED display has lower production cost as an RGB OLED display however. There are a few headsets with OLED RGB design as well, such as the Play Station VR. [101, 103–106]

3.1.3 Tethered and untethered HMDs

Whether a device is tethered or untethered is a major design choice with XR headsets. The untethered HMD's give more physical freedom for the end-user, as no cables are attached to the headset that could hinder the user's movement. The trade-off is that they also include a battery, which generally makes them more heavy and cumbersome, while also sacrificing some computing power. As the devices are limited by the energy capacity and cooling capabilities of the headset, standalone headsets often have less computational power compared to tethered headsets. The battery in untethered headsets also includes a time limit until the battery needs to be recharged.

The cooling and energy availability are key aspects when comparing a rendered software in a mobile processor chips or in PC powered tethering. The power that the computer consumes, mostly comes off as excess heat, and desktop computers consume much more power than mobile platforms. As a power comparison, a modern mobile processor spends only around a few to several watts of power in gaming [107], while a high-end PC running a high-graphics game can spend close to 500 W [108]. This gives some perspective as to why heat dissipation and power availability are bottlenecks with standalone headsets. While PC hardware can utilize large heatsinks and fans, mobile solutions, including standalone XR headsets, are limited by the weight and usability factors. A heavy headset would be more cumbersome and unwieldy for long sessions and fanning solutions for cooling have a noise factor, which limits the usability and immersion. Additionally, higher power consumption in mobile device would require a larger battery for the same use time.

3.2 Other XR solutions

Although HMDs are the most common way of achieving XR, there are other solutions available. Other immersive XR solutions include stereoscopic 3D screens, Cave Automatic Virtual Environments (CAVE), mobile (phone) AR and different physical platforms.

3.2.1 Stereoscopic screens

Stereoscopic screens have varying techniques, but they all aim at giving each eye a picture from its own perspective. Thus, the human brain can combine those pictures into a perception of a 3D object or world. An example of this is the Nintendo DS game console. The DS screen projects a different image for each eye by projecting them onto different directions. This only works when the face is at a certain distance from the display and directly in front of it however. [109]

CAVE's consist of 3 to 6 screens that are often in cubic form around the user. The screens can be rear projection screens with projectors, large displays or arrays of bezelless displays. The screens are also usually stereoscopic screens, allowing 3D scenery in every screen. To get a large cave to work with stereoscopy, the viewer's eyes or head needs to be tracked in the cave environment to ensure the screen picture adapts according to the perspective of the user. Even with the popularity of HMD's, there are multiple companies offering CAVE and similar XR solutions. In the Figure 14 is an example of a CAVE solution by IGI. The benefit that the providers market is the complete freedom of movement and the natural interaction with the system [110]. The downside of CAVEs is the high price and space requirements. [111]

Figure 14. *A CAVE solution by IGI* [112]

3.2.2 Mobile AR

Mobile AR might refer to any portable AR system, but more often it refers to AR made for mobile phones. One good example of this is the IKEA Place app, which allows the creation of a scan of a room and then trying of different IKEA products to see how they would look in the room, as seen in the Figure 15. The app also scales the IKEA products

according to the scan. Another famous AR app for smartphones is the Pokemon Go game, but there are also numerous others applications available [113]. [114]

Figure 15. *IKEA's Place app displaying a sofa in AR (Inter IKEA Systems as cited by* [114]*)*

3.3 Interaction methods

There are many ways to interact with XR systems. The most common way is to use separate controllers that often come with the various HMDs, although some headsets include integrated controls in the headset itself. The integrated controls are normal buttons, of which one could find in any electronic device. The controllers and the headset itself are tracked in orientation and often in position too.

Additional forms of interaction have become available however. Hand and eye tracking follow the user's hands or eyes and acts accordingly, voice commands obey certain key word commands, wearables are accessories that track the body part they are worn on, platforms and cockpits are elaborate, use case specific controllers, Treadmills aim to let the user move in XR in a natural way, by foot and Electroencephalography (EEG) allows the user to interact with the XR environment via brainwaves.

3.3.1 Tracking methods

As separate XR controllers are all wireless they need a tracking method for position and orientation (6 degrees of freedom or 6DOF). In addition, the headset in itself needs tracking. Some older headsets and controllers only had tracking for the orientation (3DOF), but according to the specifications of modern HMDs, 6DOF seems to be the common standard currently in headset as well as control tracking. [115]

There are effectively two different types of tracking used with XR. The inside-out method uses the cameras and sensors placed in the headset to track the surroundings, controllers and/or hands, while outside-in method uses sensors, often cameras, outside the headset to track the controllers, the user and the headset. The inside-out method is further divided into Lighthouse method and the SLAM method. [116]

In the lighthouse method, special base stations (lighthouses) are used. They sweep the room repeatedly horizontally and vertically with infrared lasers. The headsets and controllers have arrays of IR sensors and measure the time it takes between sweeps to get the position and orientation of the headset. SLAM utilizes cameras on the headset to see the surroundings and using computer vision algorithms, gyroscope and accelerometer the position of the headset can be determined. Commonly the SLAM headsets sense the controllers with separate IR cameras, as the controllers have a specific “constellation” of IR LEDs. The constellation’s orientation and position determines the position and orientation relative to the headset. [116]

The outside-in and lighthouse methods require external devices, wiring and setup, which make them more cumbersome and expensive. The lighthouse method however is considered to be the most accurate and reliable tracking method available at the moment. SLAM method is cheaper, less restrained and requires almost no setup, but loses track of the controllers/hands, if they are not in the vision of the cameras, behind the users back for instance. [116]

3.3.2 Controllers

The most common modern controllers are the HTC Vive controller, the Valve Index controller, the Oculus Touch controller and the Windows Mixed Reality (WMR) controllers. The HTC Vive controller is also called wand, for its distinctive elongated shape compared to other controllers. It is inside-out lighthouse tracked, has a USB-charged battery, grip button, trigger button, menu button, system button and a trackpad. There have been a few evolutions of these controllers with different headsets and only the VIVE Pro controller supports the latest Valve base station 2.0 tracking [117]. All of the evolutions have had the same form and functionality however. [118]

The Oculus Touch controller is an AA-battery powered, inside-out SLAM tracked controller for the Oculus quest 2 headset. Separate editions of the controller are available for different headsets and are not interchangeable, even though the form and functionality is the same in all of them. The Touch controllers have a trigger button, a grip button, a menu button, a thumbstick and 2 buttons. All the buttons and the thumbstick are also capacitive, so they sense if the user has their finger on them [119]. This allows the user to point at things in VR with the index finger or give a thumbs-up in a natural way for instance. Although the buttons do not actually sense the finger position, only if it is on the button or not. [118, 120]

Microsoft has made a reference controller design for its WMR framework for XR manufacturers to base their controllers on [121]. Samsung and HP among others have made their own controller models for WMR. Samsung odyssey+ controllers are rather close to the reference design with menu, windows, trigger and grip buttons, a trackpad and a thumbstick [122]. They have more refined ergonomics compared to the original however. HP Reverb G2 controllers on the other hand are very similar to the Oculus Touch controller in shape and ergonomics. G2 controllers also have the same buttons as the Touch controllers plus a windows button. [123]

Valve index controllers are inside-out, lighthouse tracked and have a thumbstick, trackpad with force sensor, system button, trigger button, grip force sensor, 2 buttons and accurate finger tracking. The finger tracking allows the user to do even complex hand gestures or actions and to pick up objects in a natural motion. The pressure sensors on the grip also allow the user to “squeeze” objects. The controllers are powered by USB-charged battery. [124, 125]

Of the current mainstream controllers, only the Sony Play Station VR controllers are outside-in tracked. The system works with the Play Station Camera tracking the Play Station Move or Play Station VR aim controllers. However, even Sony is moving into the direction of inside-out tracking with their oncoming VR system [126, 127]. [128]

3.3.3 Hand gestures

Hand tracking and gestures are under much research, specifically for interaction in XR. As stated by Karam [129], hands are the most suitable part of the body for human-computer interaction, even though gestures can be implemented with other limbs as well. Oculus has integrated hand tracking as an official feature in its headsets [130]. Recently, using deep-learning, Oculus has also advanced the state-of-the-art in hand tracking [131]. Their method relies on four monochromatic camera streams attached to the outer surface of the headset. For successful tracking, the user’s hands need to be in the field

of view of the outer cameras. The method provides accurate 3D hand pose estimation and runs at 60Hz on modern PC or 30Hz on the integrated mobile processor. Similar hand pose estimation methods have also been implemented by HTC in VIVE headset series. However, based on the headsets' computing power, different headset models offer different hand tracking capabilities, with the less powerful models offering simple hand position estimation (no finger tracking) and simplified gesture recognition, instead of a full hand pose estimation capability. The accuracy of hand tracking, especially when fingers/hands are occluding each other and standardization of certain gestures to specific functions are a few ongoing hot research topics. [43, 44]

Gestures can also be categorized by the implementation method used for the gesture, including wearable sensor devices, touch devices and computer vision. Touch device gestures are familiar from smartphones, while wearable and computer vision gestures are not that common in modern applications. Wearable smart watches have some gesture controls and step counters, which are often implemented by sensing swishing hands [132]. In addition, the Valve Index controllers are technically wearable hand gesture sensors, while simultaneously serving as controllers. Computer vision has been around for decades, but its use in hand gestures is still under research. [133]

Microsoft Hololens 2 has support and documentation for several hand gestures, including touch, hand ray and air tap. The gestures are implemented with computer vision. In addition to hand gestures Hololens 2 supports gaze controls with its 2 infrared cameras. [134, 135]

Leap Motion by Ultraleap is another device for accurate hand controls and is popular especially among researchers. Leap Motion is a binocular RGB camera that can be used for more accurate hand tracking than a monocular camera. There have been applications using Leap Motion on a table or attached to an HMD for hand gesture recognition. [133, 136]

3.3.4 Eye and gaze tracking

In addition to hand tracking, many applications utilize gaze tracking. This can be divided into head tracking and eye tracking. As Microsoft has designed it for their Hololens 2, with head tracking there is a pointer in the middle of the displays and it moves by turning the head. With eye tracking, no pointer is needed. By just looking at an object, the device senses what is looked at. Although eye tracking is faster, lower effort (from the user) and doesn't require a cursor, it is also not a smooth in movement (not good for drawing lines etc.) and also has difficulties with small objects. Head tracking on the other hand can provide smooth, controlled movement, is more reliable with precision and doesn't require

eye tracking hardware, which is usually expensive [106]. Gaze tracking is often used like a mouse in traditional systems, while the mouse clicks are implemented by gazing at the target for a set amount of time, with hand gestures, voice commands or controller buttons. [137]

Regarding eye tracking, most recent headsets that support this capability, usually utilize IR technologies, either integrated on the headset or as a separate extra module such as 7invensun's eye tracking module utilized in Pimax's headsets [138]. The technology used in Varjo's devices utilizes two IR cameras for each eye that operate at 100 fps with a 1280 x 800 resolution, and project a complex IR illumination pattern, advertised to result in a highly robust eye tracking system [139].

3.3.5 Voice commands

Voice is another interface that is fast popularizing with computers and XR. As virtual assistants like Siri, Google and Cortana are becoming more able and ubiquitous, speech recognition is fast evolving. Hololens 2 and WMR platform in general support specific voice commands by default. There is also an option to make custom commands in WMR, speech dictation or use Microsoft virtual assistant Cortana with speech commands. Voice commands are singular preconfigured command words mapped to specific functions, while speech dictation is meant for using speech to type text without keyboard. Magic Leap One AR glasses and Oculus headsets also support voice commands and Oculus also supports speech dictation. There are also XR platform independent softwares such as Voice Bot and Voice Attack available for voice commands and speech dictation in any PC environment, including XR environments. These allow the user to save macros for specific custom voice commands. [140–144]

3.3.6 Platforms and cockpits

Physical platforms for XR include treadmills (covered separately in 3.3.7), cockpits and some specialized solutions. An example of a specialized XR platform is the Birdly depicted in the Figure 16. Birdly is a physical VR platform for flying in VR, including a fan for wind simulation, wing control with arms and hands and a moving platform for tilt and yaw. The platform is used in conjunction with a HMD, which is responsible for the 3D audio and visuals. [145]

Figure 16. *Birdly platform in use* [146]

The VR cockpit platforms are usually very use-case specific. They range from racing seats, for racing simulator games to aeroplane cockpits for training and simulation purposes. There are also multiple do-it-yourself projects on VR seats and platforms. Many of them aim to be more general, so they can be used with car or flying simulation as needed. In the Figure 17 is an example of an aeroplane specific cockpit platform. [147–149]

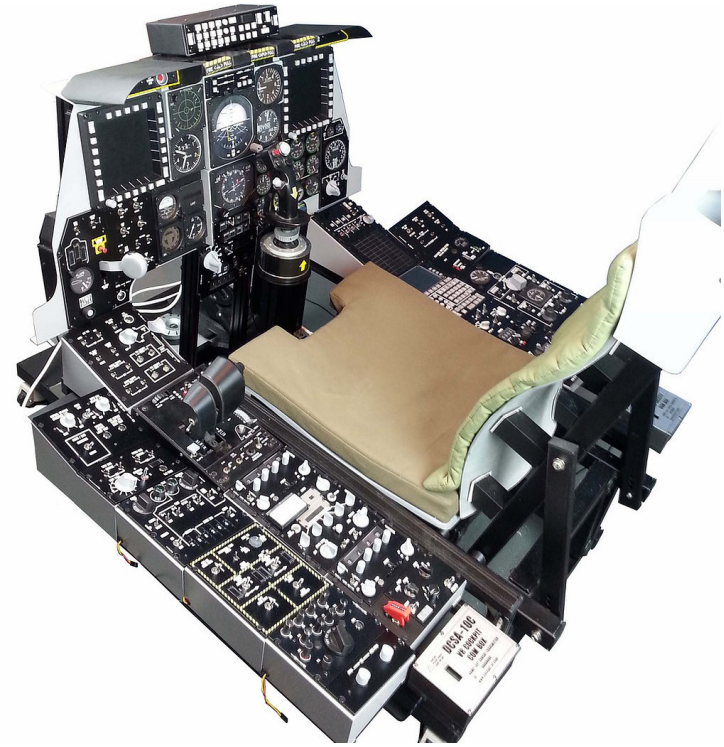

Figure 17. *A Full cockpit for VR* [147]

Platforms and cockpits allow for elaborate ways to interact with an application, but are commonly very costly compared to the default controllers.

3.3.7 Wearables

Kat-VR has made wearable set of sensors, which are designed for moving in VR. The idea is to “walk in place” to move forward in VR. The direction of movement is tied to the user’s lower body, not the direction the headset is pointing. This is done with calibration of three disk shaped sensors, one for the waist and 2 for the ankles. Another wearable solution is the kickstarter project Cybershoes. This solution involves 3 pivotal parts, (Cybershoes) that are strapped onto the feet, either without shoes or on top of shoes, a carpet (Cybercarpet) and a swiveling stool. The user sits on the stool, which is on the center of the round carpet. The cybershoes have a roller on the bottom, that senses when the user “walks in place” while sitting on the stool. The carpet ensures optimal operation with the rollers. The experience is reported to not be like walking, but rather “a step in the right direction for more immersive VR”. [150–152]

3.3.8 Treadmills

The treadmill platforms come in two variations, either a mechanical omnidirectional treadmill, like Infinadeck platform, or a low-friction platform to be used with low-friction shoes like Kat Walk, Virtuix Omni and Cyberith Virtualizer platforms. In the Figure 18 is an example of a treadmill by Virtuix. Treadmill platforms are advertised to feel more natural when moving in a VR environment and also reduce simulator sickness, which is usually involved with movement in VR environments. [150, 153–155]

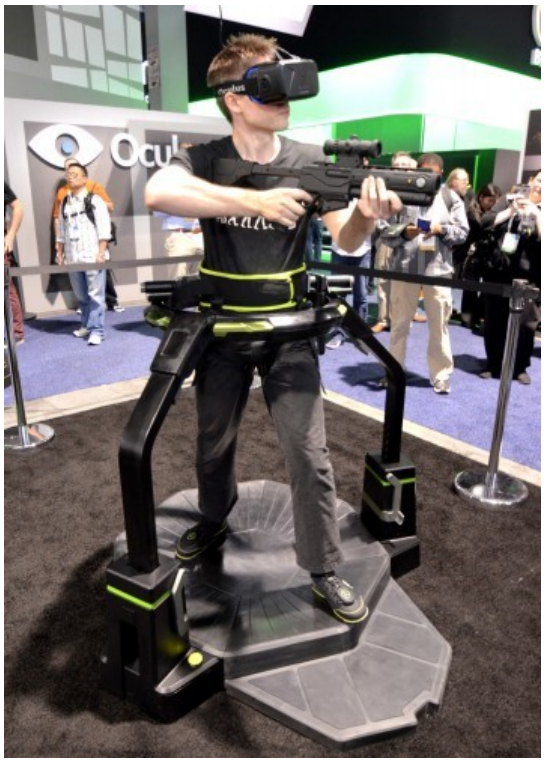

Figure 18. *Virtuix Omni treadmill* [156]

3.3.9 Electroencephalography (EEG)

There are a few accessories for XR headsets out there that claim to sense the brain activity of the user. This is done via EEG, which includes a non-invasive sensor(s) on the scalp of the user. These already allow rudimentary controls in VR environment, such as pressing a button by concentrating on the button on the screen as Nextmind has done. These buttons have a tag that is optimized for the visual cortex and can then be sensed by EEG and decoded by a computer software. Another way of interacting with an EEG device is to measure the general attentiveness or relaxation as Looxid has done. They have a few applications ready for their device that allow trying the lifting of objects by concentrating hard or checking the live EEG data on screen. What specifically needs to be concentrated on and how is not specified. [157, 158]

3.4 Feedback methods

Usually the feedback for actions with computers is auditive or visual. When a user presses a button, it induces a sound, or the button changes color, blinks or visually goes down. But there are other ways to get feedback on XR. For instance, Dextra Robotics, VRGluv, HaptX and Manus have made glove controllers, with full hand motion capture and force feedback for each individual finger. This allows getting pose information of individual fingers and gives a feeling of actually touching or holding objects in XR. The following subsections cover some other feedback methods that have been introduced into XR systems. [159–161]

3.4.1 Haptics

A step further from robotic gloves are the haptic suits provided by Teslasuit and bHaptics for instance. Both offer suits that have dozens of haptic actuators for different body areas. However, Teslasuit is designed for industrial and military sector XR with the haptics being implemented with electro-stimulation and also include biometric sensors and motion capture embedded in the suit, while bHaptics is marketed for XR gaming sector with vibrating motors for haptics and a much lower price point. [162, 163]

Another way to give haptic feedback in XR is with ultrasound, as provided by Ultraleap (previously known as Ultrahaptics). Assembling an array of ultrasound speakers, a light feedback can be induced in mid-air. This is hard to impossible to do in millimeter precision however, as for instance 40 kHz ultrasound has a wavelength of around 0.9 mm. Ultraleap promises various patterns of sensations to the palm of the hand to differentiate

between various actions. There have been many studies about using ultrasound to induce mid-air haptic feedback during the years, but it has not seen much widespread commercial application as of yet. [136, 164, 165]

3.4.2 Sound

Sounds are also an important aspect of perceived space and objects. The traditional 3D surround sound systems like 5.1 or 7.1 do not work well with XR however. The traditional model assumes fixed positions of speakers in comparison to the listener, such as front center, rear left etc., but with XR the user is able to yaw, pitch and roll their head freely, which needs to be taken into account, if an accurate immersion is to be accomplished. The research on this subject is ongoing, nevertheless there are already some providers of 3D audio plugins and solutions. However, the major breakthrough and consensus on how to do 3D audio in XR is still yet to come. [166, 167]

3.4.3 Scent

In addition to visual, auditive and haptic feedback, there are some start-up companies like OVR and Feelreal, exploring controllable scent inducing devices. The devices use an array of different scents to mimic real-life scents. The companies seem to have varying amount of scents available and most of them are attachable below commercial HMD's. The Feelreal also has a feature for spraying air or mist into the users face when activated, for instance when crossing a river in VR. [168]

4. APPLICABLE SOFTWARE FOR INTERFACING

As Unity works with C# and ROS2 works with C++ or Python by default, a software is required between them. In the sections below some available software implementations are listed that were tried. These were considered as possible solutions or bases for solutions for the interface between ROS2 and Unity.

4.1 Unity Robotics Hub

Unity Robotics Hub is an implementation of ROS1 communication in Unity by Unity Technologies. ROS2 integration for it is under development and in alpha stage at the time of writing. Enrolling in the alpha program requires signing of a non-disclosure agreement (NDA). [7]

The software was evaluated but was seen unfit for the project. The details cannot be disclosed because of the NDA.

4.2 ROS2 dotnet

ROS2-dotnet is a software developed by Esteve Fernandes along with few other contributors. It is developed for writing ROS2 applications for .NET Core and .NET Standard. This would possibly allow it to be used directly from Unity as C# is the scripting language in Unity and C# programs run on top of .NET system [169]. However, the documentation of ROS2 dotnet is very scarce and not completely up-to-date. This makes the use and learning of the software very time consuming, as significant time was invested just to get the software compiled. In addition, there is only a simple example program included with the software with ROS2 String type messages, which gives no assurance that it would work easily with more complex message types or otherwise more complex use. [74]

All these reasons combined, with emphasis on the project time constraints, made this software inadequate for the project.

4.3 ROS2 for Unity

ROS2 for unity by Dyno Robotics is another implementation of ROS2 for Unity. It uses a fork of ROS2 dotnet mentioned above. However, it doesn’t support windows and seems rather outdated with latest update to the code being over 2 years ago at the time of writing. Also based on the branch names and the documentation of the forked ROS2 dotnet, it is based on the ROS2 distro Dashing, which reached its EOL in May 2021.

The aim of the project was to incorporate ROS2 into use with VR and AR. As such, the decision was to use windows as a platform. Windows is considered by many to be more mature for XR as of yet, especially because of better graphics card and other driver availability. XR is also strongly driven by gaming industry, which is still highly concentrated around windows on PC platform. [170, 171]

These reasons combined made this software unsuitable for this project.

4.4 ROS1 implementations

ROS#, Unity Robotics Hub (without ROS2 features) and ROSBridgeLib are all implementations of ROS1 use and could be used with ROS2 via ROS1_bridge [7, 172, 173]. These would not be optimal solutions however, as ROS1_bridge brings about additional CPU stress and latency as described in the Subsection 2.3.3. It is also questionable how long is ROS1_bridge going to be supported, as the last ROS1 distro will only be supported until 2025.

4.5 Future applicability

All of the aforementioned software had pitfalls, which made them unappealing or unusable for this project. With future improvements however, they could potentially solve some or all of the problems presented in this thesis as the functionality goals of the software were well aligned with the goals of this thesis.

5. APPLICATION INTERFACE SPECIFICATION

As covered in Chapter 4, the available applicable software for interfacing ROS2 to Unity were degreed unfit and as such, a custom API was implemented. From the available official ROS2 APIs, C++ API was chosen over Python API as the baseline for the interface. This was due to Unity having official support for native library plugins made with C/C++ [174]. There also seems to be a plugin available for Unity for Python support, but it is mentioned to be made for specific cases, not directly involved with the actual game scene [175]. Outside that, Unity doesn't officially support Python, which made C++ API the only option.

The development of the ROS2-Unity library API was made largely through experimentation as the ROS2 documentation and tutorials do not cover anything more elaborate than a simple publisher, subscriber, service server, service client, action server and action client. As such the design of the ROS2-Unity library API was formed through many iterations of trying out different features of the ROS2 C++ API together with Unity. What is represented in this chapter is the final result of that iteration. The design process is covered in more detail in the Chapter 6.

The ROS2-Unity library API only has support for a handful of message types, as it was too time consuming to translate even all the ROS2 standard message types into the API. As thus, it was agreed that the API acts as a baseline to expand upon. This was supported by writing a documentation page, which describes the API in detail and gives clear instructions on how to add support for a new message type for a subscriber or publisher. The documentation describing how to add a message type to the interface for publisher can be seen in Appendix C and for subscriber in Appendix D.

The ROS2-Unity library API is composed of a low level API, a high level API and it can be expanded to ROS1 through the ROS1-Bridge. The features of these interfaces are covered in the following sections.

5.1 Low level API

The low level API consists of C style functions, as the name mangling in C++ is problematic when importing into other languages, such as C# in Unity [176]. The functions are divided into general functions and message specific functions, which are presented in the following subsections. The C++ function definitions can be seen in the Appendix A and B.

5.1.1 General functions

The ROS2-Unity library API includes the following general functions depicted in the Table 2.

Table 2. *ROS2-Unity library API: general functions*

Function name	Description
ros2_init	Initialize the library
ros2_shutdown	Shutdown library (deinitialize)
add_publisher	Adds a publisher for a given topic
add_subscriber	Adds a subscriber for a given topic

The initialize function creates 3 ROS2 nodes, one for publishing, one for subscribing and one for publishing debug messages, hard coded into the inner functions of the ROS2-Unity library. It takes one parameter of the type string, which acts as a prefix for the names of the 3 nodes created. The shutdown function releases everything the initialize function has created dynamically and it doesn't have parameters.

add_subscriber and *add_publisher* functions have 3 parameters in common, a string for the topic, an unsigned integer for the message type and an unsigned integer for backlog queue length for messages in ROS2 API. The supported message types are the ones that have a send or poll function implemented and each of the message types corresponds to a specific integer number as depicted in the Table 3. The backlog decrees how many messages are kept in queue in ROS2 API to be published or subscribed (polled or handled), before one is discarded. The *add_subscriber* function also has an unsigned integer parameter for a second backlog on the ROS2-Unity library side. This is explained in more detail in the Subsection 6.2.1.

Table 3. *ROS2-Unity library message type numbers*

Type number	Corresponding ROS2 type	Subscription support	Publisher support
0	std_msgs/msg/String	x	x
1	geometry_msgs/msg/Twist	x	x
2	std_msgs/msg/Header	x	x
3	geometry_msgs/msg/PoseWithCovarianceStamped		x
4	geometry_msgs/msg/PoseStamped		x
5	sensor_msgs/msg/Image	x	
6	sensor_msgs/msg/BatteryState	x	
7	sensor_msgs/msg/LaserScan	x	
8	sensor_msgs/msg/CompressedImage	x	
9	nav_msgs/msg/Odometry	x	

5.1.2 Message specific functions

The ROS2-Unity library API includes the following publisher functions depicted in the Table 4.

Table 4. *ROS2-Unity library API: message specific publisher functions*

Function name	Description
send_string	Sends a ROS2 String message to the given topic.
send_twist	Sends a ROS2 Twist message to the given topic.
send_header	Sends a ROS2 Header message to the given topic.
send_pose	Sends a ROS2 PoseStamped message to the given topic.
send_pose_with_cov	Sends a ROS2 PoseWithCovarianceStamped message to the given topic.

In addition the ROS2-Unity library API includes the following subscriber functions depicted in the Table 5.

Table 5. *ROS2-Unity library API: message specific subscriber functions*

Function name	Description
poll_string	Tries to poll a ROS2 String message from the given topic.
poll_twist	Tries to poll a ROS2 Twist message from the given topic.
poll_header	Tries to poll a ROS2 Header message from the given topic.
poll_image	Tries to poll a ROS2 Image message from the given topic.
poll_compressed_image	Tries to poll a ROS2 CompressedImage message from the given topic.
poll_battery_state	Tries to poll a ROS2 String message from the given topic.
poll_laser_scan	Tries to poll a ROS2 String message from the given topic.
poll_odometry	Tries to poll a ROS2 String message from the given topic.

All the publish message functions take the topic name to publish on as the first parameter followed by all the fields of the message type in question. Similarly, the poll message functions have parameters for topic name, followed by the fields of the message type in question. The message type field parameters in poll functions need to be reference parameters however, as they are updated by the function. This is how the message values

are transferred to the program calling the library functions. Polling functions for message types with strings or arrays of undefined length need to prepare a buffer for the message field data and give the length of that buffer as an input parameter. This input parameter is also given as a reference, as the library function assigns the amount of bytes written to that same variable.

ROS2 actions and services are not implemented into the ROS2-Unity library API as most of the functionality presented in this thesis can be done using only topics. In the future however, actions and services should be included into the ROS2-Unity library as well.

5.2 High level API

For easy use in Unity, a high level API for the ROS2-Unity library was introduced in Unity, the idea being to hide the low level API and the need to know about the library functions and their parameters. The high level API consists of prefabs, one for each subscription or publisher type. In the future, a universal prefab should be implemented for subscriber and publisher. This would allow the user to pick a message type and parameters directly from the Unity editor inspector tab.

5.3 ROS1 through ROS1_bridge

While ROS1_Bridge is not part of the ROS2-Unity library, it effectively expands the interface to include ROS1. As ROS1 still has a large userbase and wider array of software packages available than ROS2 at the moment, having a gateway to ROS1 networks for the ROS2-Unity library was considered important. ROS1_bridge cannot convert action calls, but as ROS2-Unity library does not utilize actions in its current configuration, this is irrelevant. [64]

ROS1_bridge was set up on a Raspberry pi 3 B+. It was able to successfully translate topics across ROS1 and ROS2. No functional failures were discovered during the use and testing of the bridge. Some performance issues were noticeable however. These issues were briefly researched and are described in more detail in the Section 6.4.

6. ROS2-UNITY LIBRARY DEVELOPMENT

This chapter covers the development process of the ROS2-Unity library. The development of the library consisted of setting up the environment, software development and software evaluation. The software development and evaluation were done in iterations, developing a feature, testing it and then moving on to the next feature. The following sections cover the aforementioned aspects of the implementation.

6.1 Environment

The primary target environment for the development consisted of Windows 10 and Unity in an x86-64 architecture computer. The secondary target environment was a tablet with Android 11 of arm64-v8a architecture. The Android build of the ROS2-Unity library was made with a cross compiler tool in an Ubuntu 18.04 environment with an x86-64 architecture computer. The ROS2 part of the environment consisted of CMake and *colcon*.

The tools of ROS2 used were rclcpp, *colcon* and the command line tools. These were all included in the ROS2 installation package. All the message definitions used in the ROS2-Unity library were also included in the installation package. The tutorials in the ROS2 documentation use CMake as the build system with the C++ examples. This was the baseline for the ROS2-Unity library as well. The development environments and compilers used are depicted in the Figure 19.

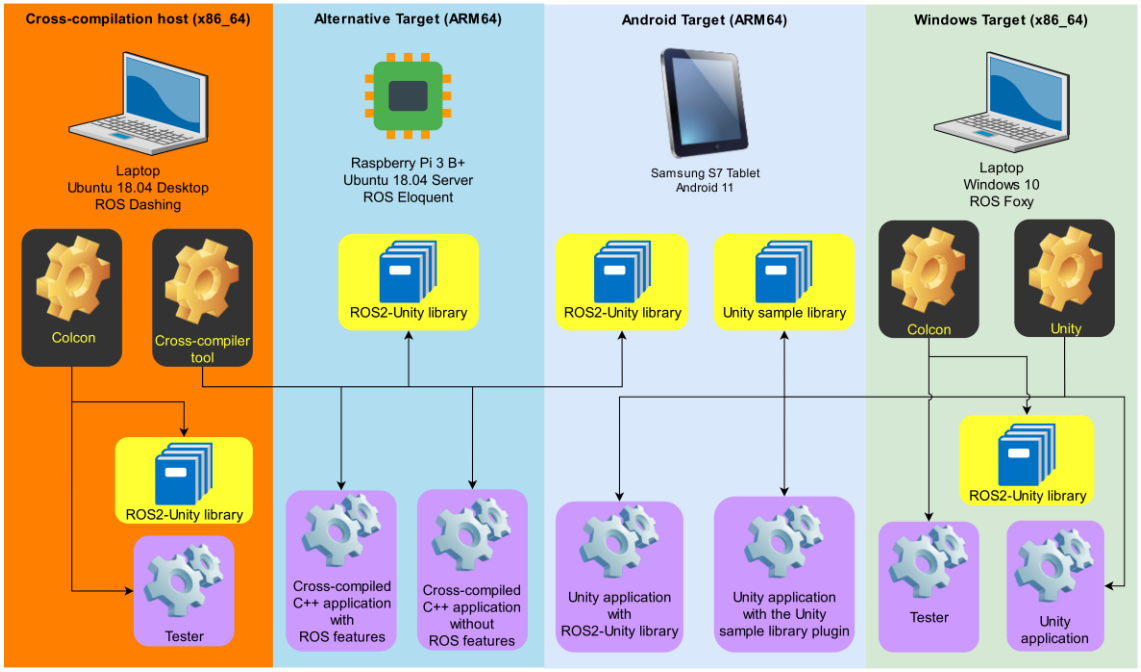

Figure 19. *The development environment for ROS2-Unity library*

6.1.1 Windows

The installation of ROS2 into Windows was made via the package manager Chocolatey, according to ROS2 documentation page instructions [13]. The process was relative simple and the instructions were clear. Compiling ROS2 software was also fairly straightforward, following the ROS2 documentation tutorials. Windows also had access to essential ROS2 tools, like RViz2 and RQt, which have been migrated from ROS1 and Ubuntu. RQt in Windows seems to have all the default plugins that RQt in Ubuntu has, except the Image View plugin, which is missing in Windows.

The used Unity editor version was 2019.4 and the SteamVR plugin was installed into it from Unity Asset Store. The SteamVR plugin came with all the necessary tools to implement rudimentary VR controls and headset integration for the demonstration scene.

6.1.2 Linux (x86-64)

The library could easily be compiled also in Ubuntu Linux environment and tested with the same C++ executable that was used for unit tests while developing in the Windows environment. There is also extensive documentation on installation and developing ROS2 on Ubuntu [13]. This platform was not a main aim of the project but was necessary for the cross-compilation attempts for the Android platform [177, 178].

6.1.3 Android

An effort was made at the end of the project to compile the ROS2-Unity library also to Android devices of arm64 architecture. The library had most of the features implemented at this stage. ROS has a cross compilation tool available, which works with ROS1 and ROS2. However, it officially only supports Ubuntu and Debian compilation targets of armhf, arm64 and x86_64 architectures. The tool also officially supports only OSX Mojave and Ubuntu 18.04 as host systems. The compilation utilizes Docker to emulate the target system and compile the software in that environment. Unity supports native library plugins for android [174]. The documentation even includes a sample Unity-package with a sample library plugin for Android. [13, 177]

6.2 Software development

This section covers the development of the ROS2-Unity library software and its utilization in Unity in a basic way as well as compiling for Android. Further development in Unity was conducted for the demonstrations, covered in Chapter 7. A simplified diagram of the ROS2-Unity library development process is depicted in the Figure 20. The iteration of adding and testing new features is covered in more detail in the Section 6.3.

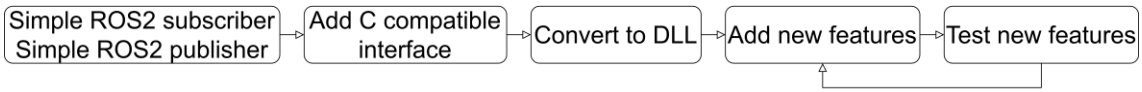

Figure 20. *ROS2-Unity library development process*

The initial research on ROS2 APIs, both python and C++, consisted of trying out the tutorials in the ROS2 documentation for writing a simple publisher and subscriber [13]. The ROS2-Unity library API was implemented as a shared library or dynamically linked library (DLL), as they are called in Windows environment [179].

The Figure 21 gives an overview of the ROS2-Unity library structure. The figure is modeled as a class diagram although only *UnityPublisher*, *UnitySubscriber* and *TopicCallback* are actual C++ classes. The other modules in the figure are considered to be packaged into modules in an object oriented way, which makes them representable as classes. *general_functions* and *overloader* are basically C wrappers for the *UnityPublisher* and *UnitySubscriber* C++ classes, while the tester module is composed of independent test functions. *TopicCallback* is a class for handling and storing messages for the subscriptions. Unity is represented here as a single class module, although a single Unity scene often consists of multiple C# classes. As the utilization of the ROS2-Unity library in Unity is not restricted to any particular classes or functions however, there is no need to specify any specific classes or functions. The same accounts to the tester module, which is covered in more detail in the Section 6.3.

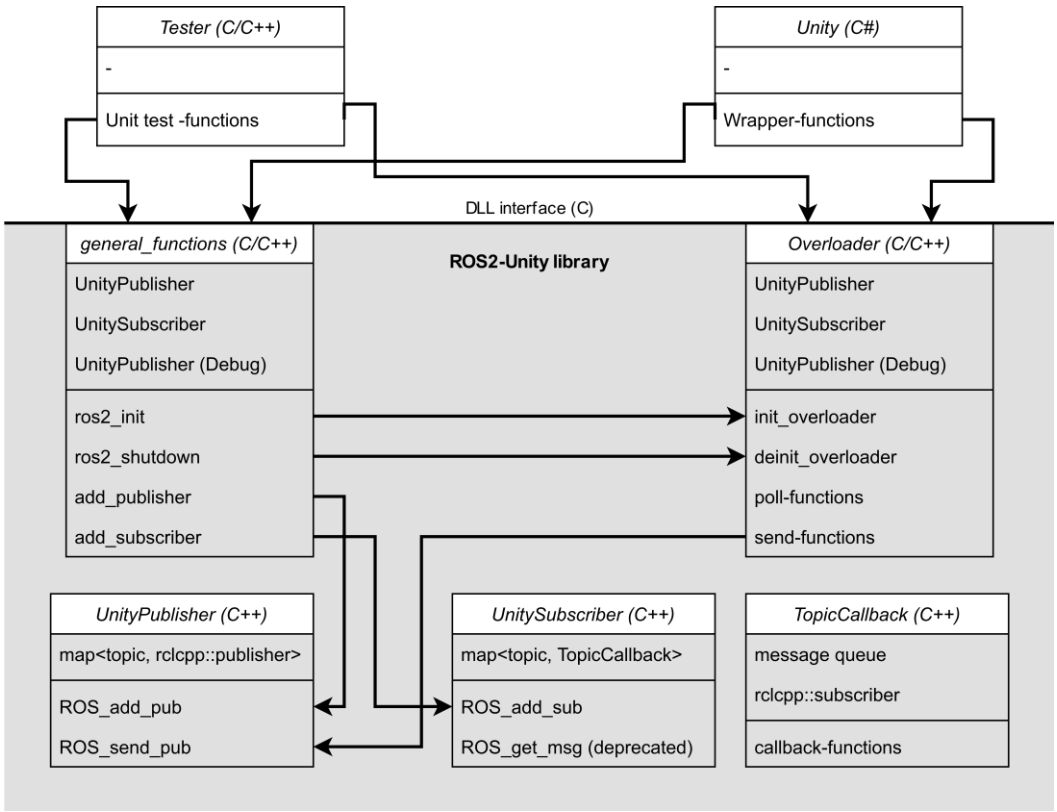

Figure 21. *ROS2-Unity library structure overview*

The following subsections describe the core functionality of ROS2-Unity library, the C compatible interface, the included message types and how the library was integrated into Unity.

6.2.1 Core functionality

The simple tutorials, of which ROS2-Unity library was built on, consisted of C++ classes for the publisher and the subscriber, which were inherited from the ROS2 node class. Both of the nodes initially only used ROS2 String messages, but they were expanded to support different message types. The tutorial publisher class had a timer, which made the node publish a message every second. The timer was removed and public methods for sending messages were introduced instead. Similarly the tutorial subscriber initially only had a callback function, which printed out the messages received. It also called the spin-function for the subscriber node, which blocks indefinitely to check messages for the subscription. [13, 17]

The callback function was modified to store the received messages in a container. Additionally a poll function was introduced, which utilizes *spin_some* function instead of regular spin function. The *spin_some* function “executes any immediately available work” [17]. In other words, it processes any available messages in queue in any of the ROS2 subscriber objects in the node and then quits. The poll function can be used to return the messages from the oldest to the newest. The spin functions are not selective of the topic or the type in question, but rather check messages on a node basis.

All the different types of messages implemented required their own add subscription method, callback function, poll message function, subscription handle and a message container. Experimentation was conducted to make the core classes as generic and reusable as possible. At this point the strong typing of C++ started to become problematic. A generic *TopicCallback* class was introduced, which could store any type of subscriber handle, message container and handler function that was integrated into the ROS2-Unity library. With this class, ROS_*add_subscription* could be made into a template function, which would only require the message type as a template parameter. Adding a subscription would create a new *TopicCallback* class and store it in a dictionary with the topic name as the key.

This was still not optimal, as all the callback functions had to be implemented manually and the containers and handlers added, when adding support for a new message type. Also as a result, the *TopicCallback* class had containers for all the other message types in addition to the one used. To alleviate the memory use, *TopicCallback* class constructor was made to shrink all the containers to a minimum size.

Similarly, the publisher class needed publisher handles, *add_publisher* functions and send message functions for every different message type. The *add_publisher* function could be made into a template function, which took the message type as a template parameter. The publisher handles were stored into separate dictionaries for all the different message types, with the topic name as the key. The message functions for different message types could be combined into one overloaded function, but the separate overloads had to be programmed manually, as finding publisher handles from the separate containers could not be generalized.

6.2.2 C compatible interface

The interface for Unity needed to be C compatible. This was achieved with the extern "C" expression, which prevents the C++ compiler from mangling the function names. The problem with the implementation was that it was built into classes and C language does not have a concept of classes. As such, separate interface functions needed to be introduced. These functions had a C compatible interface, as they were declared within extern "C" expression, the function parameters were all C compatible types and no overloading of function names was conducted. Internally however, the functions did use the C++ classes built for publishers and subscribers.

As C arrays do not monitor their size, a separate size parameter was needed for the array and string parameters of the send/poll message functions. For polling messages with string or array fields, the size variable was used both ways. From the calling side to ROS2-Unity library, the variable stated the size of the buffer in the calling side. If the buffer size was sufficient, the data was written into the array or string buffer and the size variable was assigned the amount of bytes written to the buffer or array.

This functioned well, but introduced extra steps when expanding the interface further. To include a new message type for the publisher for instance, the add publisher method under the publisher class had to be modified, send message method had to be made for the class, a new dictionary had to be created for publisher objects of the new type and interface functions to add a publisher and to send a message of the new type had to be made. In addition, if the new message type was part of a ROS2 package, which was not previously introduced to the ROS2-Unity library, it needed to be added to the package.xml manifest, to the CMakeLists.txt file and to the source files as a header.

For subscribers, the procedure was similar but instead of making a send method, a poll method was made and as an extra step, an additional callback function also had to be made. Initially the *UnitySubscriber* class also had a *ROS_get_msg* function for request-

ing received messages from the class. This would have been an extra step to also expand when adding additional message types, so it was deprecated and direct access to the class variables was used from the *overloader* poll functions. This is against the good design practice of encapsulating variables in a class, but as the schedule of the project was tight, the choice was made to save time.

Because of the steps mentioned above, it was a relatively time consuming effort to add a single message type into ROS2-Unity library. As such, only message types deemed most important were introduced into the library. In addition, a detailed documentation was created on steps to take to include a new message type for the interface. The steps can be seen in Appendix C and D for publishers and subscribers respectively. This approach allowed the initial library to be used, tested and evaluated with many standard robot functions, but also had information on how to proceed, if new message types were required in the future.

As the C interface became more bloated with the addition of multiple message types, the interface was split into *general_functions* and *overloader* modules. The *general_functions* included *ROS2_init* and *ROS2_shutdown* functions for the library in addition to the *add_publisher* and *add_subscriber* functions. The *overloader* included all the send functions for the publishers and all the poll functions for the subscribers. It was also initialized by the *ROS2_init* and deinitialized by the *ROS2_shutdown* functions in the *general_functions* module. The name *overloader* was chosen for the module as it works in a similar fashion as an overload of a function on C++. The *overloader* did not actually overload any functions however, as there is no concept of an overloaded function in the C language.

6.2.3 Included message types

The message types included in ROS2-Unity library were String, Twist, Header, PoseStamped and PoseWithCovarianceStamped for the publisher and String, Twist, Header, Image, CompressedImage, Odometry, LaserScan and BatteryState for the subscriber. This subsection describes the reasoning behind the choices of the messages included.

String messages were included from the start, but were also very important for passing on general human readable information. Future experimentation could be made on using javascript object notation (JSON) with the ROS2 String messages. This would effectively allow many kinds of data to be transmitted without altering the ROS2-Unity library interface. The downsides of JSON over more optimized message types are larger message size and inclusion of latency from serialization and deserialization of messages.

Twist includes two 3-dimensional vectors: one for linear and one for rotational speed. As such, these messages are commonly used in robots for movement, as it allows commanding the robot to move linearly in any 3D direction, as well as to rotate around any 3D axel. Moving a robot is a core functionality, which makes it imperative to include this message type. [19, 180]

Header messages include a timestamp struct with seconds and nanoseconds as subfields as well as a header ID *String* field. It is not commonly used on its own, but rather included as a part of many other messages to give sensor messages a timestamp. The purpose for including the *Header* to the interface was to experiment with transmitting non-trivial data through the interface using structs. As the *Header* message type is compact, but includes a sub-struct and a *String* field, it was ideal for this experiment. The use of structs between the ROS2-Unity library and Unity required a formation of the same struct on Unity side however, which induced unnecessary extra work. In addition the serialization and marshalling of various non-trivial structs was a non-trivial and time consuming task [181]. As such, the message fields were included directly into the interface functions rather than as structs. [19]

PoseStamped and *PoseWithCovarianceStamped* messages are used in Navigation software in ROS1, which allows a robot to navigate in a mapped environment. A *PoseStamped* message consists of a *Header*, 3D position and a quaternion orientation fields. A *PoseWithCovarianceStamped* has the same fields as the *PoseStamped* in addition to a covariance array field. The *Header* field is already described in the previous paragraph. It was possible to open the Navigation software with a map file and publish an initial pose and a goal pose using these message types through ROS1_bridge. The initial pose tells the software the position and orientation of the robot on the map, while the goal pose tells where the robot should navigate to and to which orientation. This was tested to work with a Robotnik RB1 robot in ROS1 as a ROS2 based robot was not available in the testing laboratory at the time. The *Navigation* software and the robot did end up in an undefined state many times during testing, but as this was not a core part of the research, the error was not researched further. [19, 26]

Navigation software has migrated to ROS2 with the name *Navigation 2* and it was assumed that it worked in a similar fashion as its predecessor, which is why the *PoseStamped* and *PoseWithCovarianceStamped* message types were included into the ROS2-Unity library. Unlike with the initial Navigation tool however, according to the Navigation *2* documentation, the method described above is not the used way with Navigation *2* anymore. While Navigation uses topics to send initial and goal poses, Navigation

2 uses actions. In the light of the intended purpose, this change makes the usefulness of these message types rather questionable. [20]

LaserScan message includes range, angle and intensity data of a laser scan, along with a few configuration parameters. These include a *Header*, minimum and maximum angles of the scan, the angle and time increment between measurements, time between scans and the minimum and maximum range values. As LIDAR has become more common and has many use cases in the modern society, the *LaserScan* message type was considered necessary to be included in the ROS2-Unity library [19, 182–184].

BatteryState was included as mobile robots generally work with a battery and knowing the battery state outside the robot would have many uses. The uses include programming behavioral patterns for the robot based on its battery state and human monitoring of the battery state. It was noticed that the ROS2 *BatteryState* message was quite bloated with parameters ranging from various status fields to all kinds of electrical properties of a battery. In addition, the message type supports some individual measurements for arrays of battery cells [19].

Image and *CompressedImage* were considered of high importance, as this allows manual teleoperation through camera feed or automated navigation through machine vision as well as monitoring through the robot. For the Turtlebot3 Waffle Pi, which was the robot most used in this thesis, there were a few different ROS2 camera modules with tutorials available. The used modules were *image_tools* and *v4l2_camera*. *image_tools* only allows (uncompressed) *Image* messages, while *v4l2_camera* allowed both *Image* and *CompressedImage* messages to be published. [185–187]

Odometry allows subscribing to the robots world view of position and orientation. This allows the creation of a digital twin in a virtual environment, which mimics the movement of the real robot. It is also commonly used in SLAM and navigation applications. The *Odometry* in robots is often susceptible to errors however, as discovered in the Chapter 7. [19, 20, 180]

6.2.4 Unity and ROS2-Unity library

ROS2-Unity library was imported into Unity according to the Unity documentation instructions [174]. Most of the types in C# matched with the C types used in the interface, but a few required slight changes. String buffers in C# were created as StringBuilder objects. Any pointers in the C interface were given as reference (ref) variables in C#. The arrays in the C interface, which are essentially pointers as well, were given as C# arrays.

For visualization of data, screen prefabs were created in Unity. These consisted of a vertical plane and a canvas. For textual data a text object was also created under the canvas object. For each message type, a script was created, which could be attached under a screen prefab. The scripts would then visualize the incoming/outgoing data. In addition, an initialization script was made to call *ROS2_init* function before any other calls were made to the ROS2-Unity library.

The polling of ROS2 messages in a rapid fashion induced stuttering in the Unity scene, as the calls to the ROS2-Unity library were blocking. To alleviate this, the polling on subscriber wrappers on Unity was done by creating an additional thread for it. To prevent race conditions and other undefined behavior, the calling of poll functions was protected with a mutex. In addition, a timer was introduced so messages would be polled at a constant rate.

The prefabs and scripts described were originally created as part of the integration testing, which is described in more detail in the subsection 6.3.3. While these prefabs are not adaptable to all applications as they are, they give a baseline on what to build a specific implementation upon. They also serve as an example implementation if a new publisher or subscriber script is required to be built from scratch.

To be able to publish data at will, geometric objects were created into the scene, which would create an event, if they were clicked with a mouse. This event would be caught by a publisher script, which would proceed to publish a message. As more message types and multiple screens were introduced, a need arose for movement controls in the Unity scene. For this purpose, rather traditional first person game movement controls were introduced. The player could move forward with the W key, backward with the S key, left with the A key and right with the D key (like the arrows are set up on a standard keyboard). The movement is always along the floor plane. The orientation of the player could be changed by moving the mouse while holding down the right mouse button. The controls were later expanded with robot controls for the demonstration. This is described in more detail in the Subsection 7.2.1 and depicted in the Figure 31 in the same subsection.

The XR functionality was tried with a HTC VIVE Pro HMD. The scene was tested to work the same way with the Unity XR features turned on, and the headset was tested to work while using the ROS2-Unity library. XR was only later integrated into the scene when developing demonstrations of the ROS2-Unity library with and without XR features. This is covered in the Subsection 7.2.2.

6.2.5 Cross-compiling for Android

A sample Unity scene could be successfully built for Android without any plugins. Building with the Unity sample library, consisting of only one simple function, did not work however. The Unity compiled the game program and the device ran the program successfully. The program reported an error while running however, stating that it failed to find the shared library, even though the library file was confirmed to be part of the Android application package (.apk).

To ensure the cross-compilation tool was working as intended, the ROS2-Unity library was cross-compiled to a Raspberry Pi with 64-bit Ubuntu server 18.04. The functionality was tested with a simple C++ test executable, which did not use the ROS2 features in the library, but only did simple arithmetic with an added test function. Using the ROS2 features in the library did not seem to work. The Raspberry Pi environment had Ubuntu 18.04 and ROS Eloquent, while the cross compilation environment had Ubuntu 18.04 and ROS Dashing. The mismatch between ROS2 distros might have been the reason for the error with the ROS2 functions. As the Raspberry Pi was not the primary target for cross compilation, no more work was conducted on this matter however. The main object was to check if the cross compilation tool was able to successfully compile for the arm64 architecture, which was the case.

Further research on cross-compiling libraries into Android revealed that a probable cause for a failure in platform migration of shared libraries and executables with library dependencies is the difference in the library dependencies of the programs. On a PC host system, a depended library might reside in one location in the file system, while on Android, it might reside in a different location or be absent altogether. This makes it harder to get anything but statically linked executables to work when cross-compiling from one system and architecture into a different one.

This led to a brief experimentation on compiling a statically linked library version of the ROS2-Unity library and linking that to a simple C++ program. This led to a failure as well however, as the C++ compiler used (g++) gave numerous errors when trying to compile a simple executable linked to the static library. The time constraints of the project dictated that no more effort for migrating the library to Android was to be made and the effort was declared a failure.

6.3 Software Evaluation

The ROS2-Unity library features were implemented and tested in iterative steps. The development and testing iteration is depicted in the Figure 22. This however does not

account for the initial stages of the software development process described earlier in this chapter.

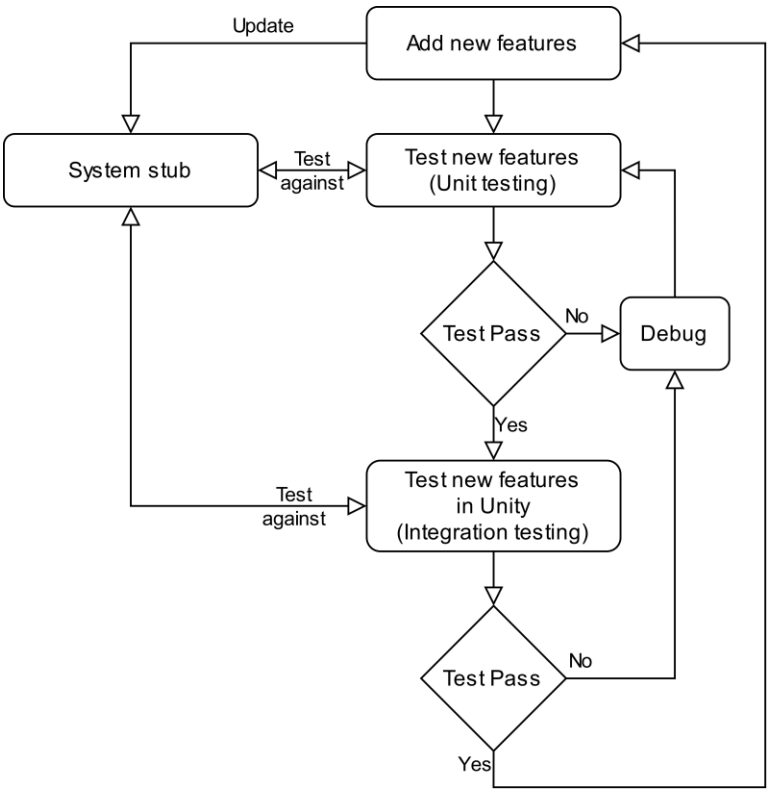

Figure 22. *Development and evaluation process*

The first step in the iteration would be to implement a new functionality, usually a new message type to subscribe or publish to. The second step would be to make or update the mock system scripts, which would act as a ROS2 system on which to test the features against. The third step would be to make a unit test for the new functionality and run it. If the test is passed, the fourth step is to copy the compiled library into a Unity project and write a script which utilizes it. The script is then tested against the same mock system script as with the unit test. This is the integration test step. If everything passes, the new functionality is added into the documentation. If any of the tests fails, the code, either in Unity or in the ROS2-Unity library, is debugged until the tests pass successfully.

The mock system scripts were ran on Ubuntu laptop, while the ROS2-Unity library, the unit tests and unity scripts were ran on a Windows laptop. The communication between systems was conducted via a WIFI LAN. The hardware setup is depicted in the Figure 23.

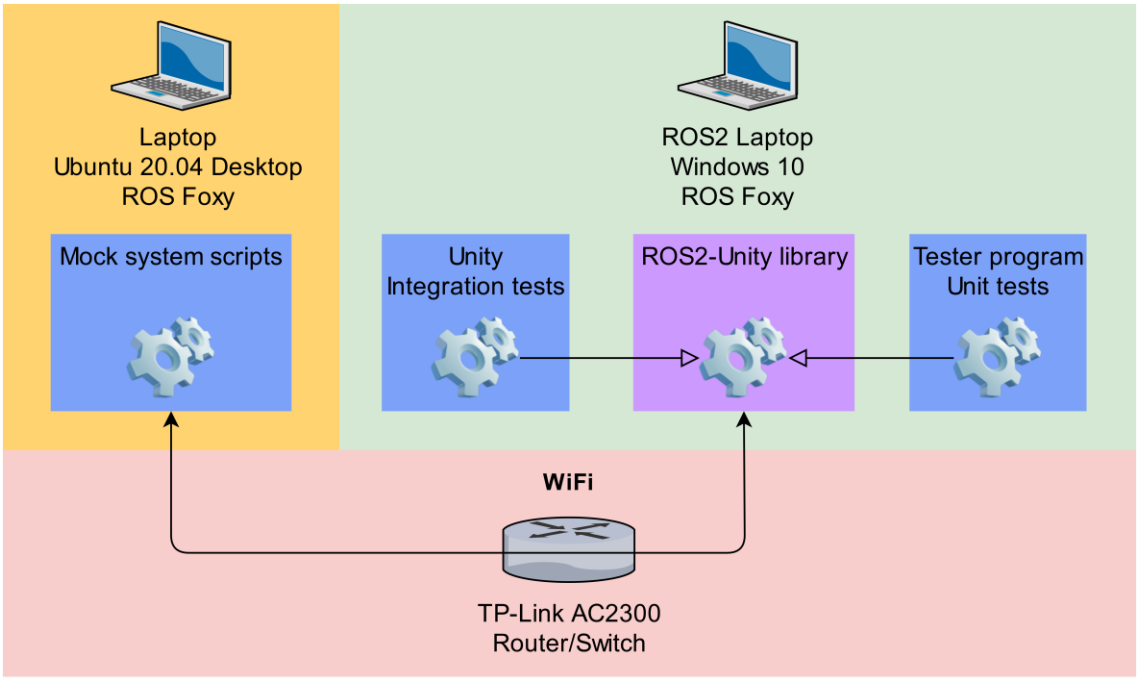

Figure 23. *ROS2-Unity library testing hardware setup*

The following subsections go into detail about the various steps and aspects of the tests.

6.3.1 Mock System scripts

2 mock system scripts were made, one for testing the publisher and one for testing the subscriber. The test scripts were written in Bash language and they utilized ROS2 command line tools.

Both of the scripts first sourced the environment for ROS2. After sourcing, the publisher script made a publisher for every type and topic that was tested by the unit and integration tests. Similarly, the subscriber script made subscribers for all the tested topics and types. As the ROS2 command line tool commands to make publishers and subscribers are blocking calls, they were made to execute in the background in a subshell and their process IDs were collected into a list. After calling the publishers, a command was called to wait for the user to input any key. This input was not stored anywhere, but acted as a signal to kill all the subprocesses that had their IDs stored previously into the list and quit. Overall in practicality, the script sent or received messages that were to be tested, until any key was pressed.

6.3.2 Unit tests

As part of the compilation process of the ROS2-Unity library software, a separate executable program was also compiled. This executable was coded in C++ and was meant as a quick way to unit test the new features of the software as well as to act as regression

tests for the older features. The features were almost exclusively new message formats for publishers or subscribers. The unit testing procedure is depicted in the Figure 24.

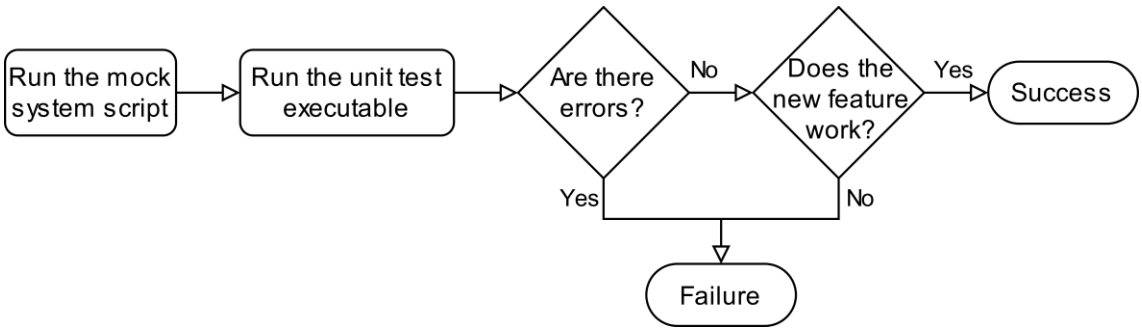

Figure 24. *Unit testing procedure*

The errors were checked from the unit test executable, while the functionality of the new feature was checked from the unit test executable or the mock system script depending on the feature. The tests were made modular, so individual features could be selected for testing or left out. The implemented test modules are listed in the Table 6.

Table 6. *Unit test modules*

General tests	Subscriber tests	Publisher tests
Base ROS2 test	Subscriber base test	Publisher base test
	Subscriber type test, including:	Publisher type test, including:
	String test	String test
	Twist test	Twist test
	Header test	Header test
	Image test	PoseStamped test
	Compressed image test	PoseWithCovarianceStamped test
	Battery state test	
	Laser scan test	
	Odometry test	

The test executable had counters for the amount of tests conducted and the amount of tests passed. For subscriber tests, there was also a separate counter for how many messages were polled and how many were received. This was to separate errors from not receiving messages, as it is not an error if no messages were received, if no messages or not enough messages were sent.

The base test module includes 3 tests. The first one initializes the library and then shuts it down, the second one initializes and shuts the library down twice and the third one initializes the library twice and then shuts the library down twice.

The Subscriber base test module also included 3 tests. The first one tests polling messages without making a subscriber. This should fail to receive a message but not crash.

The second one adds a subscription to a topic and polls it 5 times and the third one makes 5 subscribers and polls a single message from all of them.

Similarly, the Publisher base test module has 3 modules. The first one tests sending messages without making a publisher, expecting a failure but should not crash. The second one adds a publisher to a topic and sends 5 messages to it and the third one makes 5 publishers and sends a single message to all of them.

The publisher and subscriber type test modules call on an array of individual message type test modules. Each of the message type test modules initializes the library first and shuts it down in the end of the module. All of the publisher message type test modules first test if a publisher of that type can be created and then try to send 5 messages through that publisher.

Similarly, all of the subscription message type modules first test if a subscription of that type can be created and then try to poll 5 messages from that subscription. If a publisher or subscriber fails to create a publisher or a subscriber of the type in question, the test was deemed a failure. If a publisher cannot send messages to the created publisher, the test was deemed a failure. For subscribers, not receiving messages was not counted as an error, but the amount of polls and the amount of messages received by the polls were counted separately.

6.3.3 Integration tests

After a new feature was implemented and it passed the unit tests, integration testing was ensued. This evaluated the functionality of the ROS2-Unity library in a Unity environment. The integration testing procedure is depicted in the Figure 25:

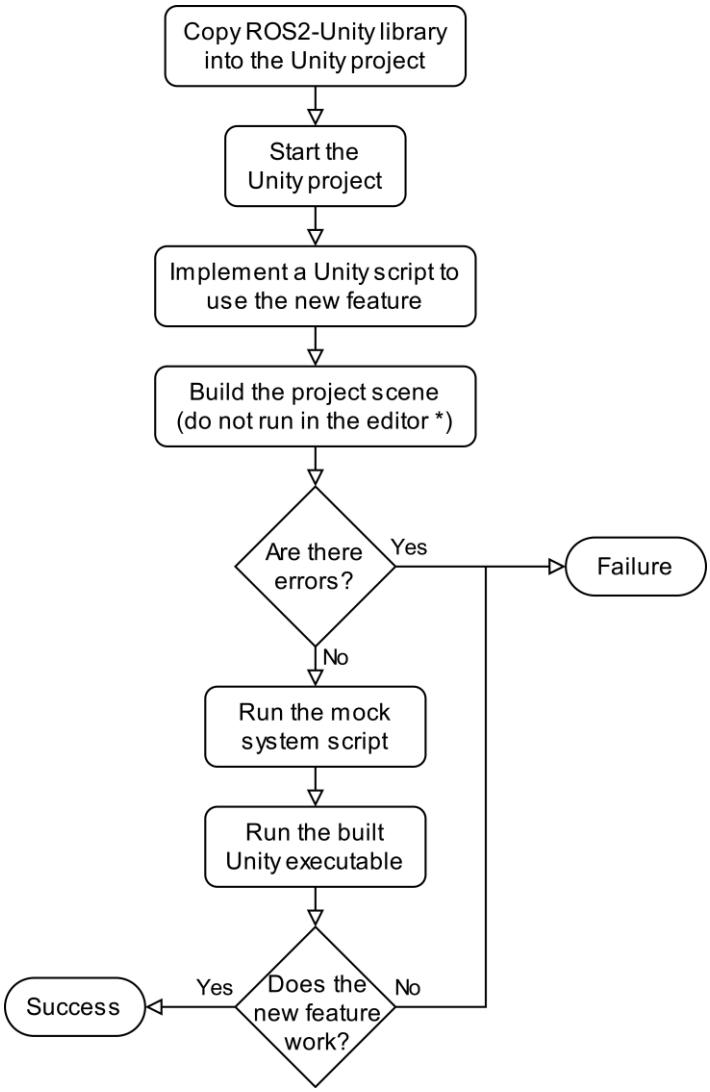

Figure 25. *Integration test procedure*

* Running the DLL on Unity editor did not work with subscriptions. In addition DLL errors on Unity editor lead to a crash, requiring a restart of the editor, which is slow.

Only errors in the Unity scene and Unity scripts could be checked in the integration tests, as DLL errors are not shown in Unity. In case of a DLL error, the program usually crashes without any given information.

The test modules had a C# script for initializing the ROS2-Unity library, a single publisher script for each of the different message types implemented and similarly a single subscriber script for each of the different message types implemented. The publisher and subscriber scripts were attached into the screen objects in the scene and showed incoming or outgoing messages on the screen. The working test modules were left as prefabs to act as a high level API for the ROS2-Unity library.

The mock system scripts worked fine with most of the message types, but for image and compressed image messages, only mock image data was sent from the mock system

script to test the core function. This was meant to be checked through a text screen in Unity to ensure the messages were received successfully. For properly testing the integration of images from ROS2 into Unity however, a Turtlebot3 with a camera was used for publishing live camera footage.

6.3.4 Stress test

Stress tests, including system lag, latency and robustness for various scenarios, was planned to be implemented only, if the project schedule allowed. Although there was no time to implement any stress testing, outside brief testing of ROS1_bridge, this is definitely an aspect, which should be researched in the future.

6.4 ROS1_bridge tests

While using ROS1_bridge, some issues were discovered. Sometimes there was a noticeable initial latency after creating a publisher or a subscriber to a topic, until the ROS1-Bridge or the recipient beyond the bridge adapted to that. In addition, while simple messages were translated seemingly instantaneously, large amounts of data induced some noticeable lag and grouping of messages. In other words, it would take some time for the messages to arrive, and when they would, there would be multiple messages arriving in bulk simultaneously.

6.4.1 Stress test

A brief test was conducted on the issue of latency and grouping of messages under stress. ROS command line tools were utilized to send large amounts of messages and the receiving of the messages was monitored through ROS command line tools for distinctive latency. In addition, Robotnik RB1 Base robot topics were subscribed to through ROS1_bridge. RB1 only has support for ROS1, and was installed with ROS Dashing at the time of testing. In addition to RB1, a PC laptop with Ubuntu 16.04 and ROS kinetic was used as ROS1 environment. The ROS2 environment it was tested against was a PC with Windows 10 and ROS Foxy. The ROS1_bridge was ran on a Raspberry Pi 3 B+ with Ubuntu server 18.04 and ROS Melodic for ROS1 and ROS Eloquent for ROS2. The communication between devices was done through a WiFi LAN with a TP-Link AC2300 router/switch. The test setup is depicted in Figure 26.

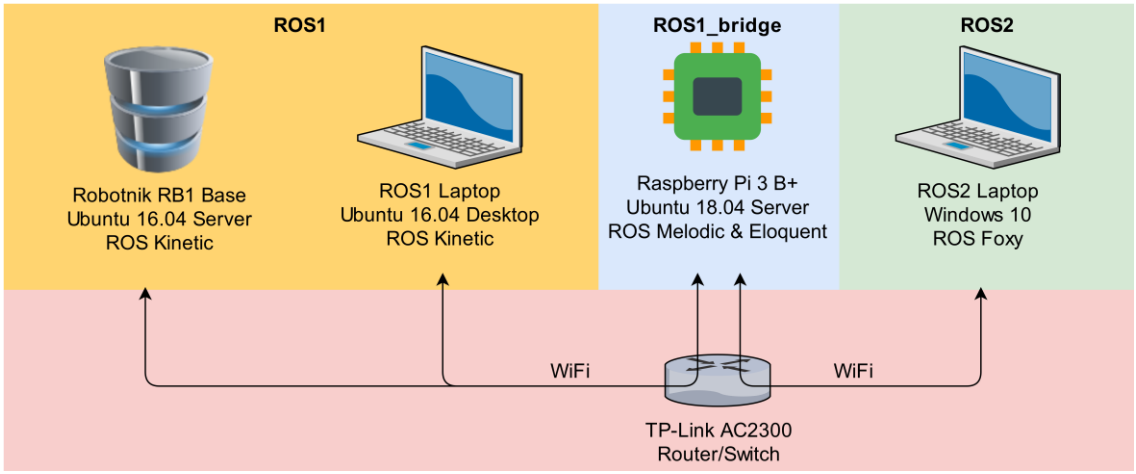

Figure 26. *ROS1_bridge stress test setup*

Trying to induce noticeable latency by sending high amounts of string messages with ROS command line tools did not work. The publishing of messages became saturated at around 5000 to 10000 Hz frequency and could not effectively be raised higher, yet there was no noticeable latency or grouping on the receiving end.

Subscribing to Robotnik RB1 Base robots /tf topic through ROS1_bridge induced a noticeable latency and grouping however. The tf package keeps track of many different coordinate frames of a robot, such as world frame, base frame and any gripper frames for instance. This can easily include a large amount of data. [26]

6.4.2 Initialization latency test

An elementary research on the initial latency with ROS1_bridge was also performed. The research was done by timing the publishing of messages both ways between ROS2 and ROS1. The demo and tutorial package executables, which come with the installation were used as the publisher and the subscriber. The timings were done by hand with a smartphone timer application. This makes the measurements relatively inaccurate, but if the measurements were consistent within each category, they could still be referenced to each other, as they were all measured in a similar manner. The test equipment and the network connections were the same as in the stress test, described in the previous subsection, excluding the RB1 robot. The setup is depicted in the Figure 27.

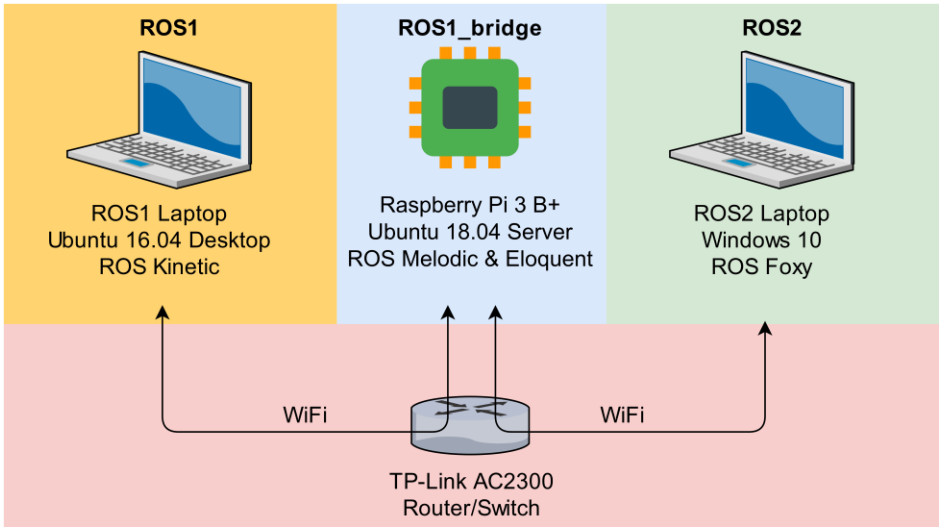

Figure 27. *ROS1_bridge initialization test setup*

The test was done by running the subscriber first and timing the first received message from the starting time of the publisher, or the other way around, running the publisher first and timing the first received message from the starting time of the subscriber. While testing, it was also noticed that messages were sometimes lost in transition when using ROS1_bridge. As in the ROS1 and ROS2 tutorial executables the messages are numbered, it is possible to check if some of the messages were lost. For cases where the publisher is initialized first, one second is given for the subscriber executable for initialization, where misses are not counted. This is considered to be the time before the actual ROS subscriber is created. As a reference, the timings of ROS2 to ROS2 and ROS1 to ROS1 messaging were also measured.

6.4.3 Results

The timings within the categories were consistent enough to be compared against other categories. They should give general information on which types of communication are the slowest and fastest to initialize through ROS1_bridge and if it needs to be taken into account when using ROS1_bridge. All of the measurements can be seen in the Appendix E. The collected time averages and average amounts of missed messages are shown in Table 7, with the missed message amounts in brackets and zero results not shown.

Table 7. *Missed message averages (in brackets with zeroes not shown) and time averages*

	Publisher first (s)	**Subscriber first (s)**
ROS1 -> ROS2	6,08	1,85
ROS2 -> ROS1	1,82	(10,7) 31,835
ROS2 -> ROS2	3,62	6,01
ROS1 -> ROS1	2,37	1,10

There were no missed messages in any of the measurements when sending across ROS1_bridge, from ROS2 to ROS2 nor from ROS1 to ROS1, except when sending from ROS2 to ROS1 and the subscriber (ROS1) was initialized first. With the publishers initialized first, messages were lost for a time period less than a second from the time the subscriber executable was launched.

The measurements show that ROS1 seems to be faster in initializing communication in general, compared to ROS2. Also considering communications through ROS1_bridge, when the ROS2 side, either publisher or subscriber, is initialized before communications, the latency is minimal. In contrast, when ROS1 side is initialized first, there is a noticeable latency. Especially with the communication from ROS2 to ROS1, when the subscriber is initialized first, there is over a half a minute latency on average. This is noticeable in the missed messages as well, as this was the only category where messages were missed. Missing over 10 seconds worth of messages after initialization on average is a high amount.

The noticeable to significant initial latency in some categories and the missing of messages suggest that an initial wait period should be included when creating a new communication topic or service across ROS1_bridge, to ensure consistent function. The measurements suggest that the wait period should be at least around 40 seconds long, as the highest delay measured was 37.14 seconds.

The tests were ran with a Raspberry Pi 3 B+ acting as the ROS1_bridge, which might have not had the computational power required for proper functionality of ROS1_bridge. A study shows that a Raspberry Pi running ROS2 has a higher latency compared to a PC [45]. Also, the ROS1_bridge on Raspberry ran on ROS Eloquent, which has reached its EOL. It is possible that the newer distros have added updates for the ROS1_bridge. The router was connected to the internet during the test, which also might have added network latency to the measurements.

To validify the magnitude of effects of ROS1_bridge in the results, a more controlled experiment and more research should be conducted in the future. One aspect to research would be to see how the timings change in more stressed messaging network. Another aspect would be to check how the ROS1_bridge functions in different systems with and without significant CPU stress. However, the test succeeded in showing categorical differences in the latency of different communication categories and allowed the making of suggestions for future research and countermeasures.

7. DEMONSTRATIONS

The demonstrations were meant as proof-of-concepts, rather than complete products. The first demonstration consisted of a Unity executable running on a large television screen and a Turtlebot3 robot in a lab environment. The executable utilizes the ROS2-Unity library to communicate with the robot through various topics.

The first live demonstration was kept without any XR involvement, as it was not necessary for demonstrating the functionality of ROS2-Unity library in a Unity environment. An XR environment and controls were implemented for the second demonstration to demonstrate the combination of XR and ROS2 however.

7.1 Setup

The VR headset used with the second demonstration was an HTC VIVE Pro with 2 HTC VIVE first generation base stations, also referred to as 1.0 base stations [188]. The default VIVE controller used in the demonstration is described in subsection 3.3.2. While the headset and the base stations do not represent the state-of-the-art of XR, they are adequate for demonstration purposes, like in this thesis.

In addition, the demonstration setup consisted of a TV screen, a Turtlebot3, a keyboard, a mouse and a laptop, which was running the Unity application with the ROS2-Unity library. The setup is depicted in the Figure 28.

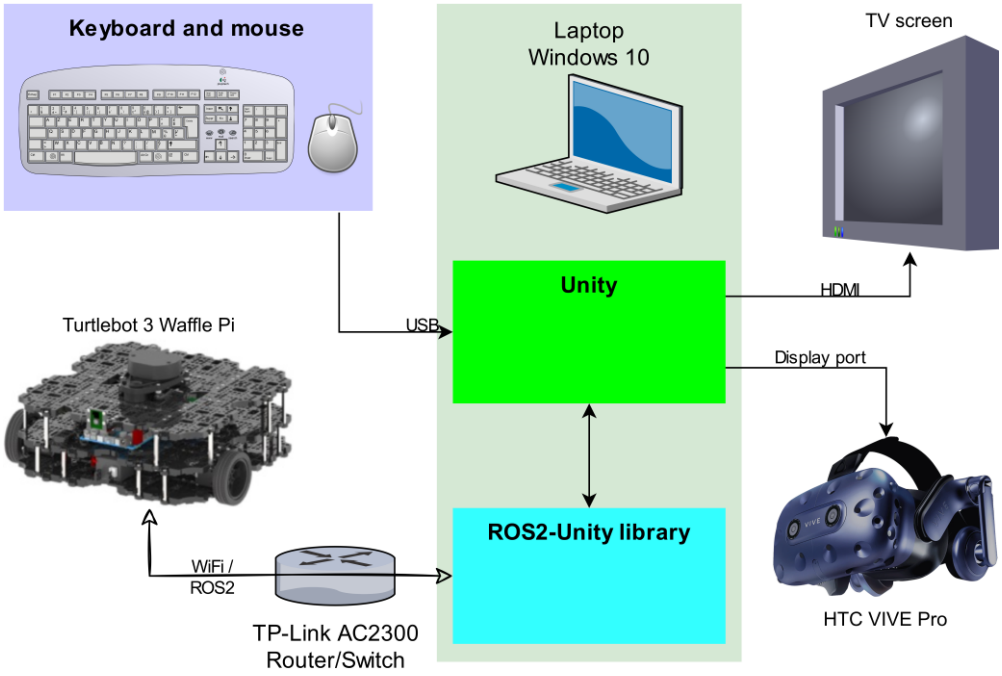

Figure 28. *The network, hardware and software setup for the demonstration*

Communication between the Turtlebot3 and the laptop was done through a WiFi LAN with a TP-Link switch/router. The Unity application was ran on the TV screen through HDMI cable and took input controls from the mouse and the keyboard, which were connected through USB. The second demonstration with XR features included had the HTC VIVE Pro HMD connected to the laptop via a Display port cable and utilized by Unity.

The player (the camera) of the Unity scene is placed inside a virtual laboratory environment, modeled after the same laboratory where the real robot was in. In the virtual laboratory, there were modeled tables and a television screen, as they were in the real lab. In addition there were 4 added information screens that were not present in the real laboratory. The screens were 2-dimensional black planes with a canvas for text on top, which could be made to show any kind of textual information. In the Figure 29 are information screens showing output commands for the robot to move (left), debug messages from the ROS2-Unity library (center) and odometry messages received from the robot (right).

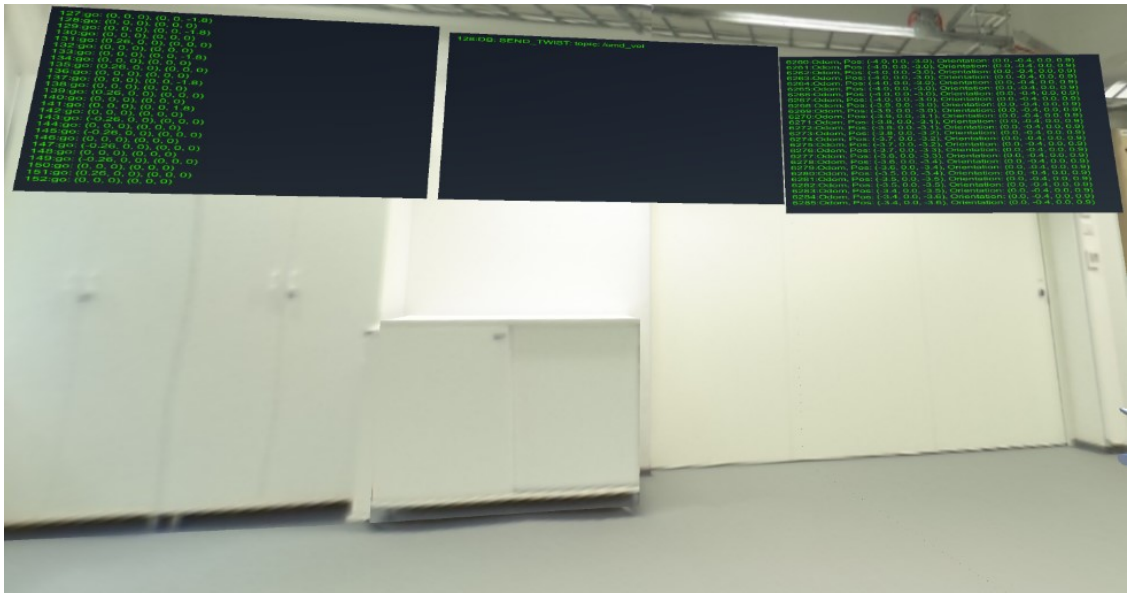

Figure 29. *Information screens in the Unity application*

In addition, there was one similar display, which displayed images, instead of text. This image display was placed onto the television screen, to look as if the images were shown by the television. This is the virtual version of the same television that the application was running on in the real world.

7.2 Features

The robot keeps track of its movements with wheel motor sensors and orientation with inertial measurement unit. This allowed the creation of a digital twin into the Unity scene by subscribing to the odometry topic of the robot (a message format that includes position and orientation of the robot [19]). A crude robot model was created into the Unity scene

and moved according to the position and orientation information of the real robot. The orientation information of the robot changes, if the robot turns itself, but also if the robot is rotated by outside force. The positional information on the other hand only changes, if the robot moves itself. The positional information can become skewed, if the robot hits an object and loses traction, as it interprets the turning wheels as linear movement in the coordinate system. Furthermore, the positional and orientational information are estimations through sensory information and can become skewed by fast and rapidly changing movements and rotations [180].

The image screen on the scene was made to show image feed from the Turtlebot3 camera through ROS2 Image message topic. The real television showing the application and the virtual television within can be seen in the Figure 30.

Figure 30. *Image screen in the Unity application*

7.2.1 Keyboard controls

In the Unity application, the player controls were the same as described in subsection 6.2.4 for the first demonstration. The mouse would control the orientation when the right mouse button was down, while W, A, S and D keys were used to move according to the floor plane. The arrow keys control the Turtlebot 3 robot by sending ROS2 *Twist* -messages. UP / DOWN arrows sent linear maximum speed (0.26 / -0.26) along x-axis, effectively making the robot move at full speed forwards or backwards respectively. LEFT / RIGHT arrows sent maximum rotational speed (1.82 / -1.82) along the z-axis, effectively turning the robot at full rotational speed counterclockwise or clockwise respectively. Values outside [-0.26, 0.26] linear and [-1.82, 1.82] rotational are dismissed by the robot.

The maximum values correspond to the reported maximum velocity values of the Turtlebot3 Waffle Pi in the official specification [180]. The keyboard controls are depicted in the Figure 31.

Figure 31. *Keyboard controls in the first demonstration*

7.2.2 XR controls

With the controller trackpad, it would have been trivial to implement similar controls to the keyboard controls described in the previous subsection. This was considered to not add any additional value to the demonstration however. As such, a different approach was taken, which would better show the capabilities of XR in conjunction with robots. The controls implemented are depicted in the Figure 32.

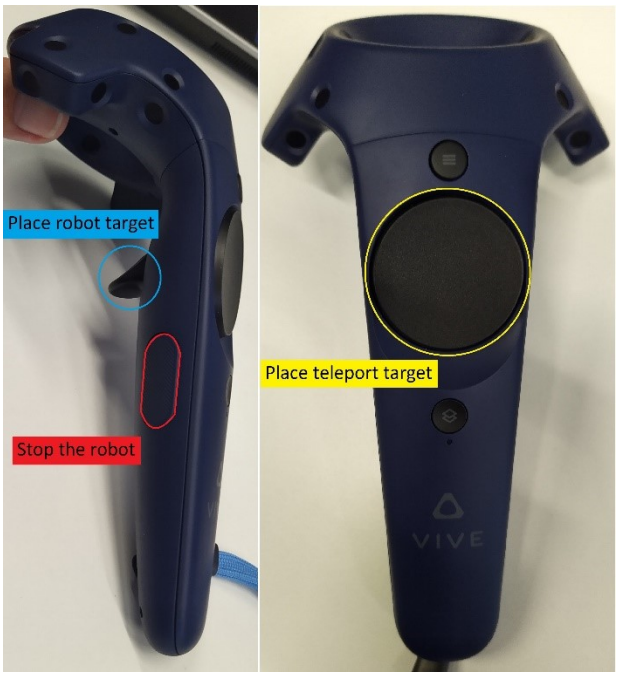

Figure 32. *XR controls mapping*

The trigger button and the touchpad of the wand controller were both made to move a target sphere in the scene to the point, where the controller was facing. This only worked on surfaces with a collider, which only included the floor and the robot itself. The trigger button placed a target for the robot, while the touchpad placed a target for the user to teleport to. The robot target was used with a simple algorithm to command the Turtlebot3 to move to the target location. The algorithm is depicted in the Figure 33. The grip button was used to stop the algorithm and the robot.

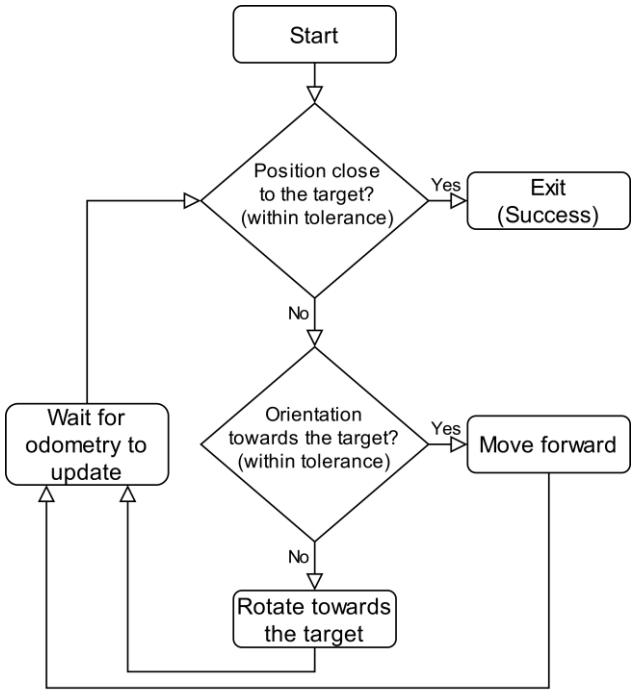

Figure 33. *Simple robot movement algorithm*

The speed and time of the movement, both rotational and linear, were configurable in the algorithm. A simple Proportional-Integral-Derivative (PID) controllers were implemented to adapt to the amount of error in the orientation and the distance from the target. As Unity received the position of the robot by subscribing to the robot's odometry topic, there was some latency between the movement of the robot and the updated odometry sent to Unity. To remedy this, the algorithm was executed in steps and a pause was introduced between the steps to let the odometry information update itself.

With the PID controllers being dependent of time between measurements for the integral and derivate parts, the pause between measurements became troublesome. The time could be measured for the movement without the pause, but with the pause, the movement with oscillation became rather slow. As such, the PID controllers were tuned to have very low integral and derivate values and the proportional values were tuned to overshoot only rarely.

7.3 Results

The first demonstration worked well in displaying the capabilities of the robot and the ROS2-Unity library. The robot was shown to receive messages from Unity application, Unity was shown to receive data from the robot and the scene showed methods for utilizing this input and output of data. The robot could be teleoperated with the keyboard. Unity application could show data received from the robot through the information screens and the image screen. With the modeled virtual environment, the robot was duplicated as a digital twin into the virtual environment and could be maneuvered with it and the image feed. The robot could be parked under a narrow space, by only using the camera feed in the Unity application. The parking of the robot can be seen in the Figure 34. The top pictures show the camera and digital twin feed in the virtual environment while parking. The bottom picture shows the end result in the real world.

Figure 34. *Parking demonstration with teleoperation and camera and position feed*

When the application had ran for about 15 minutes in the first demonstration, the participants were allowed to try out the application. At this stage, some errors started to arise. The first error froze the information screens, not displaying any new information anymore. The robot was still operatable however and the digital twin was moving in the scene according to the real robot. Restarting the application fixed this problem for about a minute, until the next error occurred. The second error prevented any information to be sent via the keyboard commands or received into the information screens. Again however, restarting the application fixed the problem, at least temporarily. The player controls in the scene were unaffected by either error.

It is unclear if these errors happened in the Unity or the ROS2-Unity library end. The first error seems to point towards unity, as the robot was still controllable after the error occurred. The second error seems to point more towards the library however, as the communication with the robot was cut, while the player controls stayed functional. More testing is required for more elaborate assumptions however.

In addition to these crashes, the robot's odometry often became skewed in long tests and in the demonstration. The Figure 35 shows a robot's digital twin positioned according to the odometry information of the real robot and a sphere, which represents the robot's starting point. When the screenshot was taken, the robot had been driven around for about 15 minutes and then driven back to the starting location and orientation. As seen in the figure, the position is off by several centimeters as the robot should stand right on top of the sphere.

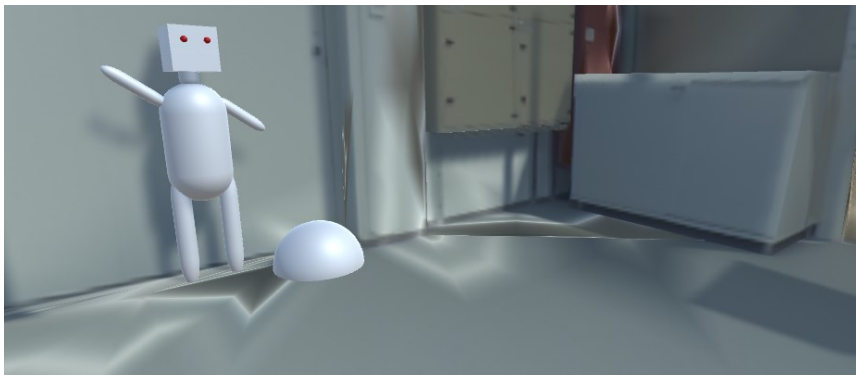

Figure 35. *A digital twin of a Robot with skewed odometry*

The second demonstration was held multiple times for singular attendees, as the VR equipment only allows for one individual to try it out at a given time. The Figure 36 depicts the demonstration.

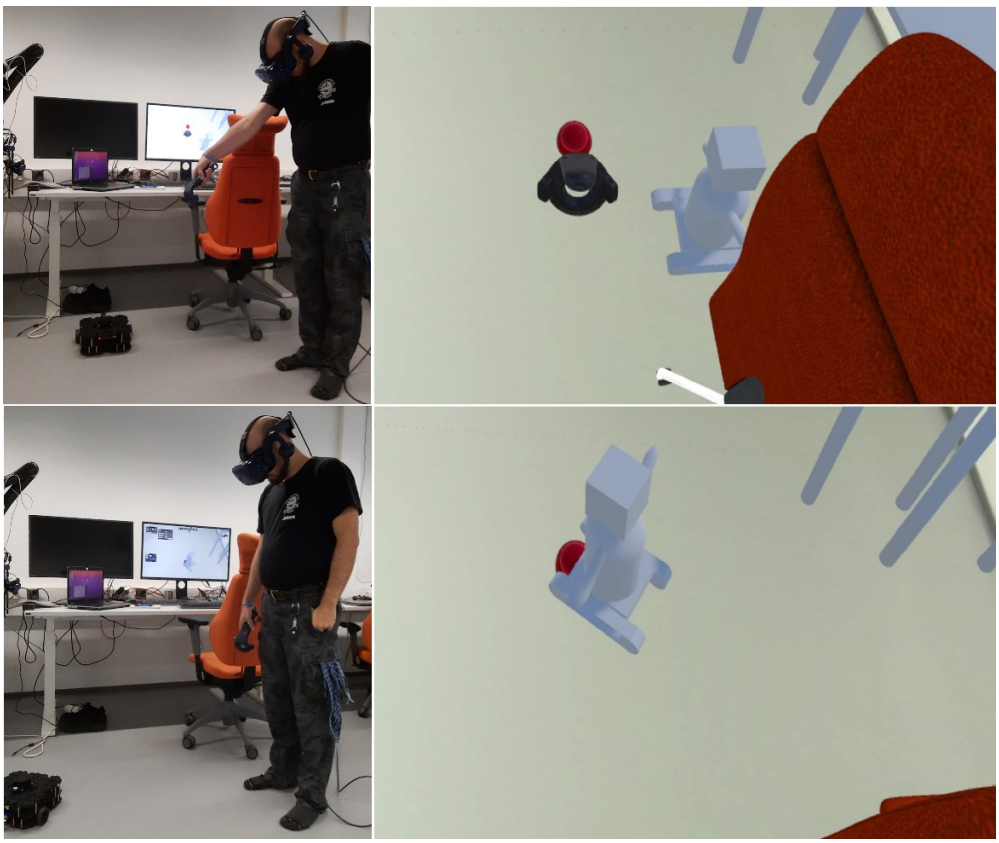

Figure 36. *ROS2-Unity library demonstration with XR.*

The left side of the figure depicts the real world, while the right side depicts the virtual world. The top side depicts giving the robot a target (the red sphere), while the bottom side depicts the robot after it has moved to the target.

The second demonstration successfully showed an example of XR-robot integration through Unity and ROS2 in a way that can be expanded upon in the future. Controlling the robot while having telepresense of the environment and having all the information screens available through teleporting in the scene were key points to the second demonstration.

The reception for the demonstrations was positive. The information screens were accessible with the teleport and keyboard controls and the robot was easily controllable through the controls. The XR demonstration successfully conveyed telepresense of the environment to the user. The setup, the algorithm and the robot were obviously not up to industry level standards with robustness, security or functionality, but presented the capabilities available with the ROS2-Unity library and XR quite well.

8. CONCLUSIONS

This study established a state-of-the-art of ROS2 and XR. Both technologies are on the rise in popularity and market value. Both are also evolving rapidly, but have yet to become widely adapted by the industry and consumers, even though both technologies and their development are also backed by powerful companies in the industry.

ROS2 has generally been agreed to be a positive change from ROS1 with improved network security, real-time support, more robust middleware design, quality-of-service and broader platform and programming language support, but the migration to ROS2 has been slow until now. The absence of the package support, lack of tool support, the learning process of the new features of ROS2 and the lack of documentation have made the adaptation unappealing in the past. This is steadily turning for the better however. Robotics companies are starting to release ROS2 drivers, the DDS suppliers are improving their product and its compatibility with ROS2 and various manufacturers of controllers, simulators and other products are adding compatibility for ROS2 into their products.

One notable pitfall in ROS2 transition is the lack of documentation however. To program anything but a trivial ROS2 program, significant research is required. Usually this means trying out functions mentioned in the ROS2 API documentation through trial and error or reading source code. This will undoubtedly slow down the adaptation of ROS2 by hobbyists and industry alike.

In XR, the HMD has stabilized as the standard way of implementing XR, even though other solutions exist. None of the other solutions, such as CAVEs and mobile AR offer the amount of immersion and the flexibility that HMDs do however. While the HMDs on the market are fast improving in display quality, also additional peripheral devices are invented and improved. The vast array of interaction and feedback methods available for XR signals of a wide interest towards the field from researchers and investors. In addition, new display types and networking solutions, such as 5G, are developed with XR as a significant target.

Notable technical problems with XR have been the induced simulator sickness and the tradeoff between the freedom of movement in untethered systems and the computational power of tethered systems. The simulator sickness has become less common due time however, as higher refresh rate and resolution displays have been constantly developed and more natural locomotion devices and designs are invented.

Various applications were tested, which could have helped or implemented an interface between ROS2 and XR through Unity. They were considered unappealing however, largely due to platform constraints and the lack of documentation, which made it difficult to use them. A custom API was implemented instead.

A functional API was created between ROS2 and XR through Unity and ROS2 C++ API. The API was named ROS2-Unity library and only consisted of ROS2 topics with select few message types. The principles for expanding the API were documented however, which should allow for more versatile functionality in the future. The message types, which were selected should allow thorough testing of basic functionalities with many different robots, without the need for expansion. Cross-compilation of the API for Android was unsuccessful however.

The functionality of the ROS2-Unity library was tested as part of the development process, however no stress testing was conducted. The ROS2-Unity library features were demonstrated with a Turtlebot3 and a Unity application. The demonstration showed the ability to control the robot through remote control and telepresense, creation of a digital twin to monitor the robot and remote monitoring through the robot camera. The Turtlebot3 did experience some errors while in use. It should be investigated if this is a problem with ROS2, ROS2-Unity library or the robot itself.

ROS1_bridge could successfully expand the library to function in ROS1 networks as well. It was also briefly stress tested as noticeable latency issues were discovered while using it. It was discovered that when creating new publishers or subscribers through ROS1_bridge, especially on the ROS2 side, it might take up to 37 seconds to start functioning. In addition, it was discovered that large amounts of data, might induce a few second latency and grouping of messages, when used through ROS1_bridge.

Additional future improvements for the ROS2-Unity library include adding ROS2 service and ROS2 action support, thoroughly testing the functionality under stress and refactoring the existing code to be more efficient and easily expanded. On the Unity side, more universal wrapper scripts should be created to give easy access to all the library functions.

In conclusion, the project was a mostly a success. The state-of-the-art of XR and ROS2 was documented and a working interface between ROS2 and Unity was achieved. Cross-compilation into Android failed however. In the future the API, or similar software, could be used in remote control of robots with telepresense through XR and governing vast sensory data through ROS2 in a virtual world as envisioned in the beginning of this project.

REFERENCES

[1] “Dictionary.com,” Available (Accessed 05.8.2021): https://www.dictionary.com/.

[2] “The Tech Terms - Computer Dictionary,” Available (Accessed 10.8.2021): https://techterms.com/.

[3] “The free dictionary,” Available (Accessed 10.8.2021): https://www.thefreedictionary.com/.

[4] “Cambridge Dictionary,” Available (Accessed 06.8.2021): https://dictionary.cambridge.org/.

[5] D. Jagneaux, “Unity 5.4 Officially Releases Streamlined VR Platform Support,” *Upload VR*, Jul. 2016, Available (Accessed 19.7.2021): https://uploadvr.com/unity-5-4-multiplatform-vr-support/.

[6] D. Heaney, “Unity Adds Toolkit For Common VR/AR Interactions,” *Upload VR*, Dec. 2019, Available (Accessed 19.7.2021): https://uploadvr.com/unity-xr-interaction-toolkit/.

[7] Unity Technologies, “Unity Robotics Hub - Github page,” Available (Accessed 31.5.2021): https://github.com/Unity-Technologies/Unity-Robotics-Hub.

[8] T. Shareef, “5 best cross-platform game engines for game developers,” *Windows Report*, Mar. 2021, Available (Accessed 19.7.2021): https://windowsreport.com/cross-platform-game-engines/.

[9] “Top 10 Best Game Engine for Beginners in 2021,” *Developer House*, Jul. 2021, Available (Accessed 19.7.2021): https://developerhouse.com/top-10-best-game-engine-for-beginners/.

[10] M. Ferguson, “5 features ROS 2 needs to add for robotics developers,” *The robot report*, Aug. 2020, Available (Accessed 07.7.2021): https://www.therobotreport.com/ros-2-5-features-robotics-developers-2020/.

[11] Open Robotics, “Why ROS2,” Available (Accessed 05.7.2021): http://docs.ros.org/en/foxy/Marketing.html.

[12] O.-N. Alejandra, AWS Events, “AWS re:Invent 2019: Change is coming to robotics development: The shift to ROS 2 (OPN201),” *YouTube*, Dec. 2019, Available (Accessed 05.7.2021): https://www.youtube.com/watch?v=kZ66pI_d5WE.

[13] Open Robotics, “ROS 2 Documentation,” Available (Accessed 05.7.2021): http://docs.ros.org/en/foxy/index.html#.

[14] “ROS.org,” Available (Accessed 05.7.2021): https://www.ros.org/.

[15] “ROS-based robot market volume worldwide between 2018 and 2024,” *Statista*, Feb. 2021, Available (Accessed 05.7.2021): https://www.statista.com/statistics/1084823/global-ros-based-robot-market-volume/.

[16] “Robot Operating System Market - Growth, Trends, COVID-19 Impact, and Forecasts (2021 - 2026) ,” *Mordor Intelligence*, 2020, Available (Accessed 05.7.2021): https://www.mordorintelligence.com/industry-reports/robot-operating-system-market.

[17] Open Source Robotics Foundation, “rclcpp documentation,” Availablehttps://docs.ros2.org/foxy/api/rclcpp/index.html.

[18] RoboJackets Training, “ROS Training - rqt,” *YouTube*, Sep. 2020, Available (Accessed 08.7.2021): https://www.youtube.com/watch?v=o90IaCRje2I.

[19] “ros2/common_interfaces - Github page,” Available (Accessed 28.6.2021): https://github.com/ros2/common_interfaces.

[20] “Navigation 2 1.0.0 documentation,” Available (Accessed 06.7.2021): https://navigation.ros.org/.

[21] “What Is SLAM (Simultaneous Localization and Mapping)?,” *MathWorks*, Available (Accessed 06.7.2021): https://www.mathworks.com/discovery/slam.html.

[22] S. Macenski, I. Jambrecic, “SLAM Toolbox: SLAM for the dynamic world,” *J. Open Source Softw.*, vol. 6, no. 61, p. 2783, May 2021, Available (Accessed 06.7.2021): https://joss.theoj.org/papers/10.21105/joss.02783.

[23] M. Steve, “SteveMacenski/slam_toolbox - Github page,” Available (Accessed 06.7.2021): https://github.com/SteveMacenski/slam_toolbox.

[24] “Gazebo - Official webpage,” Available (Accessed 07.7.2021): http://gazebosim.org/.

[25] “ROS Index - Rviz repository,” Available (Accessed 08.7.2021): https://index.ros.org/r/rviz/.

[26] “ROS Wiki,” Available (Accessed 15.7.2021): http://wiki.ros.org/.

[27] “Object Management Group - Official webpage,” *Object Management Group*, Available (Accessed 07.7.2021): https://www.omg.org/index.htm.

[28] “DDS Portal,” *DDS Foundation*, Available (Accessed 07.7.2021): https://www.dds-foundation.org.

[29] “RTI Connext DDS,” *RTI*, Available (Accessed 07.7.2021): https://www.rti.com/products.

[30] “eProsima Fast DDS,” *eProsima*, Available (Accessed 07.7.2021): https://www.eprosima.com/index.php/products-all/eprosima-fast-dds.

[31] “Eclipse Cyclone DDS,” *eclipse.org*, Available (Accessed 07.7.2021): https://projects.eclipse.org/projects/iot.cyclonedds.

[32] G. Cooperman, T. Jain, “DMTCP: Fixing the Single Point of Failure of the ROS Master DMTCP: Fixing The Single Point Of Failure Of The ROS Master,” Boston, USA, Sep. 2017, Available (Accessed 07.7.2021): https://roscon.ros.org/2017/presentations/ROSCon 2017 DMTCP.pdf.

[33] E. Knapp, N. Burek, “Quality of Service Policies for ROS2 Communications,” Oct. 2019, Available (Accessed 14.7.2021): https://roscon.ros.org/2019/.

[34] M. Asay, “How AWS is helping to open source the future of robotics,” *AWS Open Source Blog*, Oct. 2019, Available (Accessed 07.7.2021): https://aws.amazon.com/blogs/opensource/aws-helping-open-source-future-robotics/.

[35] “Sunsetting Python 2,” *Python.org*, Available (Accessed 12.7.2021): https://www.python.org/doc/sunset-python-2/.

[36] D. Thomas, “Changes between ROS 1 and ROS 2,” *ROS2 Design*, Available (Accessed 12.7.2021): http://design.ros2.org/articles/changes.html.

[37] E. Ackerman, “World Turtle Day Celebrates Final Release of ROS 1,” *IEEE Spectrum*, May 2020, Available (Accessed 07.7.2021): https://spectrum.ieee.org/automaton/robotics/robotics-software/world-turtle-day-celebrates-final-release-of-ros-1.

[38] A. Alford, “ROS 2 Foxy Fitzroy Release Improves Security and Tooling,” *InfoQ*, Jul. 2020, Available (Accessed 07.7.2021): https://www.infoq.com/news/2020/07/ros2-foxy-security-tooling/?topicPageSponsorship=1402.

[39] S. Crowe, “Open Robotics releases ROS 2 Galactic Geochelone,” *The robot report*, May 2021, Available (Accessed 07.7.2021): https://www.therobotreport.com/open-robotics-ros-2-galactic-geochelone/.

[40] N. Fragale, N. Padill, Rover Robotics, “Migrating to ROS 2 Advice from Rover Robotics,” Oct. 2019, Available (Accessed 13.7.2021): https://roscon.ros.org/2019/.

[41] G. Biggs, E. Fernandes, Autoware Foundation, “Migrating a large ROS 1 codebase to ROS 2,” Oct. 2019, Available (Accessed 13.7.2021): https://roscon.ros.org/2019/.

[42] M. Schickler, “Evaluation of ROS2 Eloquent,” *ROS Military*, Mar. 2020, Available (Accessed 05.7.2021): https://docplayer.net/209049341-Evaluation-of-ros2-eloquent.html.

[43] Rover Robotics, “ROS 2 Eloquent: Is it time to switch?,” *Rover Robotics – Blog*, Nov. 2019, Available (Accessed 13.7.2021): https://blog.roverrobotics.com/ros-2-eloquent-is-it-time-to-switch/.

[44] D. Rose, N. Fragale, “ROS 2 – Is it time to switch? [tutorial included],” *Rover Robotics – Blog*, Jul. 2019, Available (Accessed 13.7.2021): https://blog.roverrobotics.com/ros-2-is-it-time-to-switch-tutorial-included/.

[45] T. Kronauer, J. Pohlmann, M. Matthé, G. Fettweis, “Latency Analysis of ROS2 Multi-Node Systems; Latency Analysis of ROS2 Multi-Node Systems,” Jun. 2021, Available (Accessed 13.7.2021): https://arxiv.org/abs/2101.02074.

[46] M. Hansen, A. Blasdel, C. Buscaron, “ROS 2 Foxy Fitzroy: Setting a new standard for production robot development,” *AWS Robotics Blog*, Jun. 2020, Available (Accessed 14.7.2021): https://aws.amazon.com/blogs/robotics/ros2-foxy-fitzroy-robot-development/.

[47] Open Source Robotics Foundation, “rclpy documentation,” Available (Accessed 14.7.2021): https://docs.ros2.org/foxy/api/rclpy/index.html.

[48] E. Ackerman, “DARPA Awards Simulation Software Contract to Open Source Robotics Foundation,” *IEEE Spectrum*, Apr. 2012, Available (Accessed 13.7.2021):

https://spectrum.ieee.org/automaton/robotics/robotics-software/darpa-robotics-challenge-simulation-software-open-source-robotics-foundation.

[49] C. Cardoza, “Inside the Robot Operating System, the robotics industry and the Open Source Robotics Foundation,” *SD Times*, Dec. 2015, Available (Accessed 13.7.2021): https://sdtimes.com/drones/inside-the-robot-operating-system-the-robotics-industry-and-the-open-source-robotics-foundation/.

[50] “Open Robotics names ROS 2 Technical Steering Committee,” *The robot report*, Sep. 2018, Available (Accessed 07.7.2021): https://www.therobotreport.com/ros-2-steering-committee-open-robotics/.

[51] Doosan Robotics, “Doosan Robotics Unveils Industry’s First ROS Package that Supports ROS 2 Foxy Fitzroy,” *AP News*, Apr. 2021, Available (Accessed 07.7.2021): https://apnews.com/press-release/pr-newswire/technology-business-south-korea-materials-industry-robotics-511fcf63df0d36340748142a30e88319.

[52] S. Crowe, “Beta version of ROS 2 driver for UR cobots now available,” *The robot report*, May 2021, Available (Accessed 09.7.2021): https://www.therobotreport.com/beta-version-ros-2-driver-ur-cobots/.

[53] N. Martin, “Acutronic prices MARA cobot,” *MV Pro Media*, Dec. 2018, Available (Accessed 12.7.2021): https://www.mvpromedia.com/article/acutronic-robotics/.

[54] R. Tellez, “Top 10 ROS-based robotics companies to know in 2019,” *The robot report*, Jul. 2019, Available (Accessed 09.7.2021): https://www.therobotreport.com/top-10-ros-based-robotics-companies-2019/.

[55] R. Kojcev, Acutronic Robotics, “MARA, a fully Modular Articulated Robot Arm. The first ROS 2.0 based Industrial Robot”, Available (Accessed 09.7.2021): https://static1.squarespace.com/static/51df34b1e4b08840dcfd2841/t/5d102746871dcf00015ffd72/1561339746896/1_9_Kojcev.pdf.

[56] C. Coward, “New MARA Robot Arm Is Completely Modular, with ROS 2.0 Running in Every Module,” *hackster.io*, 2018, Available (Accessed 09.7.2021): https://www.hackster.io/news/new-mara-robot-arm-is-completely-modular-with-ros-2-0-running-in-every-module-6f95604ac24.

[57] E. Demaitre, “Acutronic Robotics fails to find funding for H-ROS for robot hardware,” *The robot report*, Jul. 2019, Available (Accessed 09.7.2021): https://www.therobotreport.com/acutronic-robotics-h-ros-robot-hardware-fails/.

[58] The Robot Report Staff, “NVIDIA releases Isaac simulation on Omniverse,” *The robot report*, Jun. 2021, Available (Accessed 09.7.2021): https://www.therobotreport.com/nvidia-releases-isaac-simulation-on-omniverse/.

[59] G. Andrews, “NVIDIA Isaac Sim on Omniverse Now Available in Open Beta,” *NVIDIA Developer Blog*, Jun. 2021, Available (Accessed 09.7.2021): https://developer.nvidia.com/blog/nvidia-isaac-sim-on-omniverse-now-available-in-open-beta/.

[60] “Mouser Electronics Europe,” Available (Accessed 12.7.2021): https://eu.mouser.com/.

[61] “WDL Systems,” Available (Accessed 12.7.2021): https://www.wdlsystems.com/.

[62] “ROS2 Hardware,” *ADLINK*, Available (Accessed 12.7.2021): https://www.adlinktech.com/en/ROS2-Solution.

[63] “ROScube-I from ADLINK and Intel provides ROS 2 control on the edge,” *The Robot Report*, Jun. 2020, Available (Accessed 12.7.2021): https://www.therobotreport.com/roscube-i-adliink-intel-provides-ros-2-control-edge-robots/.

[64] Open Robotics, “ros2/ros1_bridge - Github page,” Available (Accessed 29.6.2021): https://github.com/ros2/ros1_bridge.

[65] M. Zhang, “Bridging Your Transitions from ROS 1 to ROS 2,” Oct. 2019, Available (Accessed 14.7.2021): https://roscon.ros.org/2019/.

[66] R. Meertens, “Migrating Two Large Robotics ROS1 Codebases to ROS2,” *InfoQ*, Oct. 2019, Available (Accessed 12.7.2021): https://www.infoq.com/news/2019/10/migrating-ros1-ros2/.

[67] “ROS 2 - Github page,” Available (Accessed 15.7.2021): https://github.com/ros2.

[68] “rclcpp_action - Github page,” Available (Accessed 16.7.2021): https://docs.ros2.org/latest/api/rclcpp_action/.

[69] “rclada - Github page,” Available (Accessed 15.7.2021): https://github.com/ada-ros/rclada.

[70] “rcljava - Github page,” Available (Accessed 15.7.2021): https://github.com/esteve/ros2_java/tree/master/rcljava.

[71] “rclnodejs - Github page,” Available (Accessed 15.7.2021): https://github.com/RobotWebTools/rclnodejs.

[72] “rclgo - Github page,” Available (Accessed 15.7.2021): https://github.com/juaruipav/rclgo.

[73] “ros2_objc - Github page,” Available (Accessed 15.7.2021): https://github.com/esteve/ros2_objc.

[74] Fernandez Esteve, “ros2-dotnet - Github page,” Available (Accessed 31.5.2021): https://github.com/ros2-dotnet/ros2_dotnet.

[75] “ros2_rust - Github page,” Available (Accessed 15.7.2021): https://github.com/ros2-rust/ros2_rust.

[76] D. Thomas, “Colcon - Universal Build Tool,” Oct. 2019, Available (Accessed 15.7.2021): https://roscon.ros.org/2019/.

[77] “colcon documentation,” Available (Accessed 15.7.2021): https://colcon.readthedocs.io/en/released/#.

[78] J. Pomerantz, “Extending XR across Campus: Year 2 of the EDUCAUSE/HP Campus of the Future Project,” *Educause*, May 2020, Available (Accessed 30.6.2021): https://www.educause.edu/ecar/research-publications/extending-xr-across-campus-year-2-of-the-educause-hp-campus-of-the-future-project/appendices.

[79] “What is AR, VR, MR, XR, 360?,” *Unity*, Available (Accessed 30.6.2021): https://unity3d.com/what-is-xr-glossary.

[80] P. Milgram, H. Takemura, A. Utsumi, F. Kishino, “Augmented reality: a class of displays on the reality-virtuality continuum,” in *Proceedings of SPIE - The International Society for Optical Engineering*, Jan. 1994, vol. 2351, Available (Accessed 30.6.2021): https://www.researchgate.net/publication/228537162_Augmented_reality_A_class_of_displays_on_the_reality-virtuality_continuum.

[81] Business Finland, “Mixed Reality solutions from Finland,” 2021, Available (Accessed 29.6.2021): https://www.businessfinland.fi/49962f/globalassets/finnish-customers/news/news/2021/mixed-reality-solutions-from-finland-offering.pdf.

[82] IDC, “Worldwide Semiannual Augmented and Virtual Reality Spending Guide,” 2019,

[83] Accenture, G20 Young Entrepreneurs Alliance, “WAKING UP TO A NEW REALITY Building a responsible future for immersive technologies,” May 2019, Available (Accessed 30.6.2021): https://www.accenture.com/sg-en/insights/technology/responsible-immersive-technologies.

[84] “Why Airlines are Embracing xR Services,” *NEC*, Available (Accessed 30.6.2021): https://www.nec.com/en/global/insights/article/2020022507/index.html.

[85] D. Le Jehan, “How XR is helping enterprise to reap the benefit of end of distance,” *Telegraph*, Jul. 2018, Available (Accessed 30.6.2021): https://www.telegraph.co.uk/business/essential-insights/business-benefits-of-xr/.

[86] K. Stanney, B. D. Lawson, B. Rokers, M. Dennison, C. Fidopiastis, T. Stoffregen, S. Weech, J. M. Fulvio, “Identifying Causes of and Solutions for Cybersickness in Immersive Technology: Reformulation of a Research and Development Agenda,” *Int. J. Hum. Comput. Interact.*, vol. 36, no. 19, pp. 1783–1803, Nov. 2020, Available (Accessed 27.7.2021): https://doi.org/10.1080/10447318.2020.1828535.

[87] E. Kidwell, “How to avoid motion sickness caused by VR headsets while gaming,” *Windows Central*, Apr. 2018, Available (Accessed 30.6.2021): https://www.windowscentral.com/how-avoid-motion-sickness-caused-by-vr-headsets-while-gaming.

[88] S. Weech, T. Wall, M. Barnett-Cowan, “Reduction of cybersickness during and immediately following noisy galvanic vestibular stimulation,” *Exp. Brain Res.*, vol. 238, no. 2, pp. 427–437, Jan. 2020, Available (Accessed 27.7.2021): https://link.springer.com/article/10.1007/s00221-019-05718-5.

[89] D. Saredakis, A. Szpak, B. Birckhead, H. A. D. Keage, A. Rizzo, T. Loetscher, “Factors associated with virtual reality sickness in head-mounted displays: A systematic review and meta-analysis,” *Front. Hum. Neurosci.*, vol. 14, p. 96, 2020, Available (Accessed 30.6.2021): https://www.ncbi.nlm.nih.gov/pmc/articles/PMC7145389/.

[90] “VR and AR pushing connectivity limits,” Oct. 2018, Available (Accessed 30.6.2021): https://www.qualcomm.com/media/documents/files/vr-and-ar-pushing-connectivity-limits.pdf.

[91] T. Alsop, “Global market spend on XR technologies by industry 2018-2023,” *Statista*, Feb. 2021, Available (Accessed 29.6.2021): https://www.statista.com/statistics/1096765/global-market-spend-on-xr-technologies-by-industry/.

[92] Mordor Intelligence, “Extended Reality (XR) Market - Growth, Trends, COVID-19 Impact, and Forecasts (2021 - 2026),” Available (Accessed 17.5.2021): https://www.mordorintelligence.com/industry-reports/extended-reality-xr-market.

[93] B. Marr, “The 5 Biggest Virtual And Augmented Reality Trends In 2020 Everyone Should Know About,” *Forbes*, Jan. 2020, Available (Accessed 30.6.2021): https://www.forbes.com/sites/bernardmarr/2020/01/24/the-5-biggest-virtual-and-augmented-reality-trends-in-2020-everyone-should-know-about/?sh=323bc9c224a8.

[94] Qualcomm, “Qualcomm Collaborates with 15 Global Operators to Deliver XR Viewers,” May 2020, Available (Accessed 30.6.2021): https://www.qualcomm.com/news/releases/2020/05/26/qualcomm-collaborates-15-global-operators-deliver-xr-viewers.

[95] B. Lang, “How to Tell if Your PC is VR Ready,” *Road to VR*, May 2021, Available (Accessed 30.6.2021): https://www.roadtovr.com/how-to-tell-pc-virtual-reality-vr-oculus-rift-htc-vive-steam-vr-compatibility-tool/.

[96] “VIVE EU - Official webpage,” Available (Accessed 29.6.2021): https://www.vive.com/eu/.

[97] “Rent The Hololens 2,” *Hartford Technology Rental*, Available (Accessed 05.8.2021): https://hartfordrents.com/product/hololens-2-rental/.

[98] P. Kourtesis, S. Collina, L. A. A. Doumas, S. E. MacPherson, “Technological Competence Is a Pre-condition for Effective Implementation of Virtual Reality Head Mounted Displays in Human Neuroscience: A Technological Review and Meta-Analysis,” *Frontiers in Human Neuroscience*, vol. 13,Frontiers Media S.A., p. 342, Oct. 2019, Available (Accessed 30.6.2021): https://www.ncbi.nlm.nih.gov/pmc/articles/PMC6783565/.

[99] I. Goradia Ȧ, J. Doshi Ȧ, L. A. Kurup Ȧ, “A Review Paper on Oculus Rift & Project Morpheus,” *Rev. Artic. Int. J. Curr. Eng. Technol.*, vol. 4, no. 5, Oct. 2014, Available (Accessed 10.8.2021): http://inpressco.com/category/ijcet.

[100] Valve Corporation, “Field of View,” Available (Accessed 29.6.2021): https://www.valvesoftware.com/en/index/deep-dive/fov.

[101] M. Rakver, “VR Display Comparison – LCD vs OLED, (Sub)Pixels, FOV & PPD,” *Smart Glasses Hub*, Apr. 2020, Available (Accessed 30.6.2021): https://smartglasseshub.com/vr-headset-display-comparison/.

[102] A. Frich, “The basics or color in color management guide,” Apr. 2015, Available (Accessed 30.6.2021): https://www.color-management-guide.com/introduction-color-management.html.

[103] OLED-Info, “Pentile OLEDs: introduction and market status,” Feb. 2019, Available (Accessed 29.6.2021): https://www.oled-info.com/pentile.

[104] Kore, “Display technologies for Augmented and Virtual Reality - Inborn Experience (UX in AR/VR),” *Medium*, Oct. 2018, Available (Accessed 30.6.2021):

https://medium.com/inborn-experience/isplay-technologies-for-augmented-and-virtual-reality-82feca4e909f.

[105] Intel, “Everything You Need to Know About Gaming Monitors,” Available (Accessed 30.6.2021): https://www.intel.com/content/www/us/en/gaming/resources/everything-you-need-to-know-about-gaming-monitors.html.

[106] R. Brown, “VRcompare,” Available (Accessed 30.6.2021): https://vr-compare.com/.

[107] S. Segan, “Snapdragon 855 Gives a Big Boost to Phones’ Batteries,” *PC magazine*, Dec. 2018, Available (Accessed 02.8.2021): https://www.pcmag.com/news/snapdragon-855-gives-a-big-boost-to-phones-batteries.

[108] E. Mills, N. Bourassa, L. Rainer, J. Mai, A. Shehabi, N. Mills, “Green Gaming: Energy Efficiency without Performance Compromise,” Sep. 2018, Available (Accessed 30.6.2021): https://docs.google.com/a/lbl.gov/viewer?a=v&pid=sites&srcid=bGJsLmdvdnxncmVlbmdhbWluZ3xneDo0ZDBmMTI1MTViZTViODY5.

[109] N. Oxford, “How Does the 3D Effect Work on the Nintendo 3DS?,” *Lifewire*, Mar. 2020, Available (Accessed 30.6.2021): https://www.lifewire.com/nintendo-3ds-3d-images-1126263.

[110] “Virtual Reality,” *Viscon*, Available (Accessed 30.6.2021): https://viscon.de/en/vr-2/vr-cave/.

[111] A. Tarbi, “VR CAVE system: an immersive technology,” *Laval Virtual*, Aug. 2020, Available (Accessed 30.6.2021): https://blog.laval-virtual.com/en/vr-cave-system-an-immersive-technology/.

[112] “IGI - Official webpage,” Available (Accessed 29.6.2021): https://www.werigi.com/.

[113] T. Kataja, “2D Animation in the World of Augmented Reality,” May 2019, Available (Accessed 30.6.2021): https://www.theseus.fi/bitstream/handle/10024/172563/Kataja_Terhikki.pdf?sequence=2&isAllowed=y.

[114] A. Ayoubi, “IKEA Launches Augmented Reality Application,” *Architect Magazine*, Sep. 2017, Available (Accessed 30.6.2021): https://www.architectmagazine.com/technology/ikea-launches-augmented-reality-application_o.

[115] D. Barnard, “Degrees of Freedom (DoF): 3-DoF vs 6-DoF for VR Headset Selection,” *VirtualSpeech*, May 2019, Available (Accessed 30.6.2021): https://virtualspeech.com/blog/degrees-of-freedom-vr.

[116] D. Heaney, “How virtual reality positional tracking works,” *VentureBeat*, May 2019, Available (Accessed 30.6.2021): https://venturebeat.com/2019/05/05/how-virtual-reality-positional-tracking-works/.

[117] Valve Corporation, “Valve Index -base station,” *Steam*, Available (Accessed 01.7.2021): https://store.steampowered.com/app/1059570/Valve_Index_tukiasema/.

[118] Cas and Chary VR, “Valve Index Controllers VS Oculus Touch VS HTC VIVE Controllers (Review),” *YouTube*, Aug. 2019, Available (Accessed 30.6.2021): https://www.youtube.com/watch?v=LPVvn3RzSVU.

[119] “Oculus Controller Input Mapping,” *Oculus Developers*, Available (Accessed 30.6.2021): https://developer.oculus.com/documentation/unreal/unreal-controller-input-mapping-reference/.

[120] “Oculus,” Available (Accessed 17.5.2021): https://www.oculus.com/.

[121] J. Feltham, “Microsoft’s 6DOF Controllers Are A Reference Design, Not A Product,” *UploadVR*, Jul. 2017, Available (Accessed 30.6.2021): https://uploadvr.com/microsofts-6dof-controllers-reference-design-not-product/.

[122] “All You Need to Know About the HMD Odyssey Controllers,” *Samsung*, Available (Accessed 01.7.2021): https://www.samsung.com/us/support/answer/ANS00078171/.

[123] C. Hunt, “What type of motion controllers is the HP Reverb G2 using?,” *Windows Central*, Jun. 2020, Available (Accessed 01.7.2021): https://www.windowscentral.com/hp-reverb-g2-controllers.

[124] A. Robertson, “The Valve Index might have the most fun VR controllers I’ve ever tried,” *The Verge*, May 2019, Available (Accessed 30.6.2021): https://www.theverge.com/2019/5/28/18639084/valve-index-steamvr-headset-knuckles-controllers-preview.

[125] H. Dingman, “Valve Index Controllers review: These revolutionary VR controllers need patience to hit full potential,” *PCWorld*, Jun. 2019, Available (Accessed 30.6.2021): https://www.pcworld.com/article/3405879/valve-index-controllers-review.html.

[126] H. Nishino, “Next-gen VR on PS5: the new controller,” *PlayStation.Blog*, Mar. 2021, Available (Accessed 01.7.2021): https://blog.playstation.com/2021/03/18/next-gen-vr-on-ps5-the-new-controller/.

[127] S. Barker, “PSVR’s Next-Gen Motion Controllers Tracked by Headset,” *Push Square*, Mar. 2021, Available (Accessed 01.7.2021): https://www.pushsquare.com/news/2021/03/psvrs_next-gen_motion_controllers_tracked_by_headset.

[128] Sony Interactive Entertainment Inc., “PS VR, PS Camera, Move Controllers,” *Republic of Communications*, Available (Accessed 30.6.2021): https://news.cision.com/fi/republic-of-communications/i/ps-vr--ps-camera--move-controllers,c2563635.

[129] M. Karam, “A framework for research and design of gesture-based human-computer interactions,” University of Southampton, 2006, Available (Accessed 30.6.2021): https://eprints.soton.ac.uk/263149/.

[130] A. Robertson, “Oculus Quest games are getting controller-free hand tracking this month,” *The Verge*, May 2020, Available (Accessed 01.7.2021): https://www.theverge.com/2020/5/18/21260554/oculus-quest-anniversary-hand-tracking-third-party-games-beat-saber-tracks.

[131] B. Smith, C. Wu, H. Wen, P. Peluse, Y. Sheikh, J. K. Hodgins, T. Shiratori, “Constraining Dense Hand Surface Tracking with Elasticity,” *Facebook Research*, Dec. 2020, Available (Accessed 30.6.2021): https://research.fb.com/publications/constraining-dense-hand-surface-tracking-with-elasticity/.

[132] V. Genovese, A. Mannini, A. M. Sabatini, “A Smartwatch Step Counter for Slow and Intermittent Ambulation,” *IEEE Access*, vol. 5, pp. 13028–13037, May 2017, Available (Accessed 01.7.2021): https://www.researchgate.net/publication/316903839_A_Smartwatch_Step_Counter_for_Slow_and_Intermittent_Ambulation.

[133] Y. LI, J. HUANG, F. TIAN, H.-A. WANG, G.-Z. DAI, “Gesture interaction in virtual reality,” *Virtual Real. Intell. Hardw.*, vol. 1, no. 1, pp. 84–112, Feb. 2019, Available (Accessed 30.6.2021): https://www.sciencedirect.com/science/article/pii/S2096579619300075#f6.

[134] Microsoft, “Gaze and commit - Microsoft Docs,” Oct. 2019, Available (Accessed 29.6.2021): https://docs.microsoft.com/en-us/windows/mixed-reality/design/gaze-and-commit.

[135] Microsoft, “HoloLens 2 - Microsoft official webpage,” Available (Accessed 29.6.2021): https://www.microsoft.com/en-us/hololens/hardware.

[136] “Ultraleap - Official webpage,” Available (Accessed 17.5.2021): https://www.ultraleap.com/.

[137] “Gaze and commit | Microsoft Docs,” Available (Accessed 17.5.2021): https://docs.microsoft.com/en-us/windows/mixed-reality/design/gaze-and-commit.

[138] S. Hayden, “Pimax’s Long-awaited Eye & Hand Tracking Modules Shipping to Backers ‘in a month,’” *Road to VR*, Jun. 2020, Available (Accessed 02.7.2021): https://www.roadtovr.com/pimaxs-eye-hand-tracking-release-date/.

[139] “Varjo - Official webpage,” Available (Accessed 29.6.2021): https://varjo.com/.

[140] “Voice input -Microsoft Docs,” Oct. 2019, Available (Accessed 30.6.2021): https://docs.microsoft.com/en-us/windows/mixed-reality/design/voice-input.

[141] A. Strange, “This Is How Magic Leap’s Voice Control Works via the New Lumin OS Update,” *Magic Leap :: Next Reality*, Dec. 2019, Available (Accessed 30.6.2021): https://magic-leap.reality.news/news/hands-on-is-magic-leaps-voice-control-works-via-new-lumin-os-update-0217220/.

[142] “Voice commands and dictation - Oculus support,” Available (Accessed 30.6.2021): https://support.oculus.com/3287153321370720/.

[143] “VoiceBot,” *Binary Fortress Software*, Available (Accessed 30.6.2021): https://www.voicebot.net/.

[144] “VoiceAttack - Official webpage,” Available (Accessed 29.6.2021): https://voiceattack.com/.

[145] “Birdly VR,” Available (Accessed 17.5.2021): https://birdlyvr.com/.

[146] A. Lilly, “Fulfilling the Dream of Human Flight with Virtual Reality,” *AIXR*, Jun. 2020, Available (Accessed 30.6.2021): https://aixr.org/insights/human-flight-virtual-reality/.

[147] “Cockpit-VR,” Available (Accessed 17.5.2021): https://www.cockpit-vr.com/.

[148] “Next Level Racing,” Available (Accessed 17.5.2021): https://nextlevelracing.com/.

[149] C. Hall, “With Star Wars: Squadrons on the way, it’s time to build the cockpit of your dreams,” *Polygon*, Jun. 2020, Available (Accessed 30.6.2021): https://www.polygon.com/guides/2020/7/1/21300566/how-to-build-a-flight-simulator-racing-cockpit-homemade-diy-rig-frame-chair-setup.

[150] “KAT VR,” Available (Accessed 17.5.2021): https://www.kat-vr.com/.

[151] E. Switzer, “Cybershoes For Oculus Quest Review: Two Steps Forward, One Step Back,” *The Gamer*, Jan. 2021, Available (Accessed 30.6.2021): https://www.thegamer.com/cybershoes-for-oculus-quest-review/.

[152] D. Nield, “Cybershoes could solve the problem of walking around in VR,” *New Atlas*, Nov. 2020, Available (Accessed 30.6.2021): https://newatlas.com/vr/cybershoes-virtual-reality-walking/.

[153] “Cyberith,” Available (Accessed 17.5.2021): https://www.cyberith.com/.

[154] “Infinadeck,” Available (Accessed 17.5.2021): https://www.infinadeck.com/.

[155] “Virtuix - Official webpage,” Available (Accessed 29.6.2021): https://www.virtuix.com/.

[156] B. Lang, “Virtuix Exploring Crowdfunded Equity Investment Under US ‘JOBS Act,’” *Road to VR*, Jan. 2016, Available (Accessed 30.6.2021): https://www.roadtovr.com/virtuix-exploring-crowdfunded-equity-investment-under-us-jobs-act/.

[157] “NextMind,” Available (Accessed 17.5.2021): https://www.next-mind.com/.

[158] “LooxidLabs,” Available (Accessed 17.5.2021): https://looxidlabs.com/.

[159] “Dexta Robotics,” Available (Accessed 17.5.2021): https://www.dextarobotics.com/.

[160] “VRgluv,” Available (Accessed 17.5.2021): https://www.vrgluv.com/enterprise.

[161] “HaptX,” Available (Accessed 17.5.2021): https://haptx.com/.

[162] BHaptics, “TactSuit,” Available (Accessed 17.5.2021): https://www.bhaptics.com/.

[163] “TESLASUIT,” Available (Accessed 17.5.2021): https://teslasuit.io/.

[164] T. Carter, S. A. Seah, B. Long, B. Drinkwater, S. Subramanian, “UltraHaptics: Multi-point mid-air haptic feedback for touch surfaces,” in *UIST 2013 - Proceedings of the 26th Annual ACM Symposium on User Interface Software and Technology*, 2013,

pp. 505–514, Available (Accessed 30.6.2021): http://dx.doi.org/10.1145/2501988.2502018.

[165] I. Rakkolainen, A. Sand, R. Raisamo, “A Survey of Mid-Air Ultrasonic Tactile Feedback,” *2019 IEEE Int. Symp. Multimed.*, 2019, Available (Accessed 30.6.2021): https://ieeexplore.ieee.org/stamp/stamp.jsp?arnumber=8959023&tag=1.

[166] E. Wesemann, “Why 3D audio is the next big step for virtual reality,” *VentureBeat*, Jul. 2017, Available (Accessed 30.6.2021): https://venturebeat.com/2017/07/21/3d-audio-is-the-next-big-step-for-virtual-reality/.

[167] M. Johansson, “VR for Your Ears: Dynamic 3D Audio Is Coming Soon,” *IEEE Spectrum*, Jan. 2019, Available (Accessed 30.6.2021): https://spectrum.ieee.org/consumer-electronics/audiovideo/vr-for-your-ears-dynamic-3d-audio-is-coming-soon.

[168] J. Kalish, “Can Smell-O-Vision Save VR?,” *PC magazine*, Dec. 2019, Available (Accessed 02.7.2021): https://uk.pcmag.com/news/124081/can-smell-o-vision-save-vr.

[169] “A Tour of the C# language,” *Microsoft Docs*, Jan. 2021, Available (Accessed 10.8.2021): https://docs.microsoft.com/en-us/dotnet/csharp/tour-of-csharp/.

[170] Patola, Boiling Steam, “The State of Virtual Reality on Linux,” Available (Accessed 31.5.2021): https://boilingsteam.com/the-state-of-virtual-reality-on-linux/.

[171] Nitish S., “Linux Gaming vs. Windows Gaming,” Available (Accessed 31.5.2021): https://www.fosslinux.com/45763/linux-gaming-vs-windows-gaming.htm.

[172] Siemens, “ROS# - Github page,” Available (Accessed 19.7.2021): https://github.com/siemens/ros-sharp.

[173] “ROSBridgeLib - Github page,” Available (Accessed 19.7.2021): https://github.com/srv/ROSBridgeLib.

[174] “Unity Documentation - Official webpage,” Available (Accessed 19.7.2021): https://docs.unity3d.com/Manual/index.html.

[175] “Unity documentation - Python for Unity,” Available (Accessed 19.7.2021): https://docs.unity3d.com/Packages/com.unity.scripting.python@2.0/manual/index.html.

[176] “Name mangling (C++ only) - IBM Documentation,” Available (Accessed 19.7.2021): https://www.ibm.com/docs/en/i/7.2?topic=linkage-name-mangling-c-only.

[177] K. Emerson, “ros-tooling/cross_compile - Github page,” Available (Accessed 24.6.2021): https://github.com/ros-tooling/cross_compile.

[178] Open Robotics, “ROS 2 Documentation - Cross-compilation,” Available (Accessed 24.6.2021): https://docs.ros.org/en/foxy/Guides/Cross-compilation.html.

[179] J. Whitham, “Differences between dynamically linked libraries (.dll) and shared objects (.so),” *Blog of Jack Whitham*, Oct. 2017, Available (Accessed 21.7.2021): https://www.jwhitham.org/2017/10/dll.html.

[180] Robotis, “TurtleBot3 eManual,” Available (Accessed 28.6.2021): https://emanual.robotis.com/docs/en/platform/turtlebot3/features/.

[181] “Marshaling Classes, Structures, and Unions,” *Microsoft Docs*, Mar. 2017, Available (Accessed 26.7.2021): https://docs.microsoft.com/en-us/dotnet/framework/interop/marshaling-classes-structures-and-unions.

[182] X. Wang, H. Pan, K. Guo, X. Yang, S. Luo, “The evolution of LiDAR and its application in high precision measurement,” *IOP Conf. Ser. Earth Environ. Sci.*, vol. 502, no. 1, Jun. 2020, Available (Accessed 21.7.2021): https://iopscience.iop.org/article/10.1088/1755-1315/502/1/012008.

[183] “What is Lidar and what is it used for?,” *American Geosciences Institute*, Available (Accessed 21.7.2021): https://www.americangeosciences.org/critical-issues/faq/what-lidar-and-what-it-used.

[184] E. Borneman, “Mapping the Entire Surface of the Earth with LiDAR,” *Geography Realm*, Feb. 2020, Available (Accessed 30.6.2021): https://www.geographyrealm.com/mapping-the-entire-surface-of-the-earth-with-lidar/.

[185] S. van Dijk, “Raspberry Pi + ROS 2 + Camera,” *Medium*, Nov. 2020, Available (Accessed 26.7.2021): https://medium.com/swlh/raspberry-pi-ros-2-camera-eef8f8b94304.

[186] “ROS Index - image_tools repository,” Available (Accessed 26.7.2021): https://index.ros.org/p/image_tools/github-ros2-demos/#foxy.

[187] “ROS Index - v4l2_camera repository,” Available (Accessed 26.7.2021): https://index.ros.org/r/v4l2_camera/#foxy.

[188] I. Hamilton, “Valve Releases New Details About Next Generation Tracking Technology,” *Upload VR*, Oct. 2017, Available (Accessed 20.7.2021): https://uploadvr.com/valve-new-base-station-info/.

APPENDIX A: GENERAL_FUNCTIONS.H

```
#ifndef ROS2UNITYDLL_H
#define ROS2UNITYDLL_H


#ifdef DLL_EXPORT
#define ROS_C_API __declspec(dllexport)
#else
#define ROS_C_API __declspec(dllimport)
#endif


#include <iostream>
#include <string>
#include <stdlib.h>
#include <stdio.h>
#include <sstream>

#include "UnitySubscriber.h"
#include "UnityPublisher.h"
#include "overloader.h"

#include "rclcpp/rclcpp.hpp"
#include "rclcpp/node.hpp"
#include "std_msgs/msg/string.hpp"
#include "sensor_msgs/msg/image.hpp"
#include "sensor_msgs/msg/compressed_image.hpp"
#include "sensor_msgs/msg/battery_state.hpp"
#include "sensor_msgs/msg/laser_scan.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include "geometry_msgs/msg/pose_stamped.hpp"
#include "geometry_msgs/msg/pose_with_covariance_stamped.hpp"
#include "nav_msgs/msg/odometry.hpp"


extern "C" {
    // Common ROS functions
    ROS_C_API void ros2_init(const char *node_name_prefix);

    ROS_C_API void ros2_shutdown();


      /*
    topic: topic name. Needs to be lowercase
    type: number for a corresponding type of message
        0: string
    backlog: amount of messages to have in queue
    ret: true: success, false: fail        */
    ROS_C_API bool add_publisher(const char *topic,
                                 unsigned int type,
                                 unsigned int ROS_backlog);
```

```
      /*
    topic: topic name. Needs to be lowercase
    type: number for a corresponding type of message
    backlog: amount of messages to have in queue
    ret: true: success, false: fail    */
    ROS_C_API bool add_subscriber(  const char *topic,
                                          unsigned int type,
                                          unsigned int DLL_backlog,
                                          unsigned int ROS_backlog);
}

#endif
```

APPENDIX B: OVERLOADER.H

```
#ifndef OVERLOADER_H
#define OVERLOADER_H


#ifdef DLL_EXPORT
#define ROS_C_API __declspec(dllexport)
#else
#define ROS_C_API __declspec(dllimport)
#endif

#include <iostream>
#include <string>
#include <stdlib.h>
#include <stdio.h>
#include <cstdint>

#include "UnitySubscriber.h"
#include "UnityPublisher.h"

#include "std_msgs/msg/string.hpp"
#include "sensor_msgs/msg/image.hpp"
#include "sensor_msgs/msg/compressed_image.hpp"
#include "sensor_msgs/msg/battery_state.hpp"
#include "sensor_msgs/msg/laser_scan.hpp"
#include "geometry_msgs/msg/twist.hpp"
#include "geometry_msgs/msg/pose_stamped.hpp"
#include "geometry_msgs/msg/pose_with_covariance_stamped.hpp"
#include "nav_msgs/msg/odometry.hpp"


bool init_overloader(   shared_ptr<UnityROSSubscriber> sub,
                        shared_ptr<UnityROSPublisher> pub,
                        shared_ptr<UnityROSPublisher> err);

bool deinit_overloader();


extern "C" {
    // -----  Subscriber  -----

    /*
    topic: topic name
    out_buf: buffer to write the message into
    buf_size: the size of the buffer on input,
            amount of variables written on output          */
    ROS_C_API void poll_string(     char *topic,
                                    char *out_buf,
                                    size_t *buf_size);
```

```
/*
    topic: topic name
    double x, y, z: linear vector
    double rx, ry, rz: angular vector               */
    ROS_C_API void poll_twist(      char *topic,
                                    double *x,
                                    double *y,
                                    double *z,
                                    double *rx,
                                    double *ry,
                                    double *rz);

    /*
    topic: topic name
    unsigned int sec, nanosec: timestamp
    frame_id: string
    buf_size: the size of the buffer on input,
            amount of variables written on output        */
    ROS_C_API void poll_header(     char *topic,
                                    uint32_t *sec,
                                    uint32_t *nanosec,
                                    char *frame_id,
                                    size_t *buf_size);


/*
    topic: topic name
    unsigned int sec, nanosec: timestamp
    frame_id: string
    frame_id_size: size of frame_id buffer on input,
            amount of variables written on output.
    height: image height in pixels
    width: image width in pixels
    encoding: color encoding used, 'rgb8' for
            instance (16 bytes for buffer are enough)
    is_bigendian: boolean
    step: amount of bytes per row (width * bytes_per_pixel.
            with rgb8: width * 3)
    data: image data
    data_buf_size: size of data buffer on input,
            amount of variables written on output.
    NOTE: If buffer is too small or no message or topic was found,
            frame_id_size and data_buf_size will be 0.           */
    ROS_C_API void poll_image(      char *topic,
                                    uint32_t *sec,
                                    uint32_t *nanosec,
                                    char *frame_id,
                                    size_t *frame_id_size,
                                    uint32_t *height,
                                    uint32_t *width,
                                    char *encoding,
                                    uint8_t *is_bigendian,
                                    uint32_t *step,
                                    uint8_t *data,
                                    size_t *data_buf_size);
```

```
/*
    topic: topic name
    unsigned int sec, nanosec: timestamp
    frame_id: string
    frame_id_size: size of frame_id buffer on input, bytes written on output
    float voltage, temperature, current, charge,
    float capacity, design capacity, percentage
    power_supply_status: 0-4, see ros2 message documentation for more info
    power_supply_health: 0-8, see ros2 message documentation for more info
    power_supply_technology: 0-6, see ros2 message documentation for more
info
    present: boolean, is a battery present
    float[] cell_voltage, cell_temperature: for battery cell arrays
    cell_amount: cell_voltage and cell_temperature buffer lengths on input,
            amount of values written on output.
            If 0 on input, no data will be written
    location: string
    location_size: buffer size on input, amount written on output
    serial_number: string
    serial_size: buffer size on input, amount written on output            */
    ROS_C_API void poll_battery_state(    char *topic,
                                          uint32_t *sec,
                                          uint32_t *nanosec,
                                          char *frame_id,
                                          size_t *frame_id_size,
                                          float *voltage,
                                          float *temperature,
                                          float *current,
                                          float *charge,
                                          float *capacity,
                                          float *design_capacity,
                                          float *percentage,
                                          uint8_t *power_supply_status,
                                          uint8_t *power_supply_health,
                                          uint8_t *power_supply_technology,
                                          bool *present,
                                          float *cell_voltage,
                                          float *cell_temperature,
                                          size_t *cell_amount,
                                          char *location,
                                          size_t *location_size,
                                          char *serial_number,
                                          size_t *serial_size
                                        );
```

```
/*
    topic: topic name
    unsigned int sec, nanosec: timestamp
    frame_id: string
    frame_id_size: size of frame_id buffer on input, bytes written on output
    float angle_min, angle_max, angle_increment,
      float time_increment, scan_time, range_min, range_max,
    ranges: float[]
    ranges_size: buffer size on input, amount of values written on output
    intensities: float[]
    intensities_size: buffer size on input, amount of values written on out-
put.
            If device doesn't support this == 0
    */
    ROS_C_API void poll_laser_scan( char *topic,
                                    uint32_t *sec,
                                    uint32_t *nanosec,
                                    char *frame_id,
                                    size_t *frame_id_size,
                                    float *angle_min,
                                    float *angle_max,
                                    float *angle_increment,
                                    float *time_increment,
                                    float *scan_time,
                                    float *range_min,
                                    float *range_max,
                                    float *ranges,
                                    size_t *ranges_size,
                                    float *intensities,
                                    size_t *intensities_size);


    /*
    topic: topic name
    unsigned int sec, nanosec: timestamp
    frame_id: string
    frame_id_size: size of frame_id buffer on input,
            amount of variables written on output.
    format: the picture format. "jpeg" or "png" accepted officially
            but not guaranteed. Prepare for 16 bytes.
    data: image data
    data_buf_size: size of data buffer on input, amount of variables
            written on output.
    NOTE: If buffer is too small or no message or topic was found,
            frame_id_size and data_buf_size will be 0.                    */
    ROS_C_API void poll_compressed_image( char *topic,
                                          uint32_t *sec,
                                          uint32_t *nanosec,
                                          char *frame_id,
                                          size_t *frame_id_size,
                                          char *format,
                                          uint8_t *data,
                                          size_t *data_buf_size);
```

```
/*
    topic: topic name
    unsigned int sec, nanosec: timestamp
    frame_id: string
    frame_id_size: size of frame_id buffer on input,
            amount of variables written on output.
    child_frame_id: Frame id the pose points to.
            The twist is in this coordinate frame. (from ros2 github)
    child_frame_id_size: size of child_frame_id buffer on input,
            amount of variables written on output.
    no_covariance: input bool if covariance values are wanted on output
    x, y, z, quat_x, quat_y, quat_z, quat_w: position and orientation (Pose)
    vel_x, vel_y, vel_z, rot_x, rot_y, rot_z: linear and rotational speed
    covariance: pose covariance
    vel_covariance: velocity covariance                */
    ROS_C_API void poll_odometry(   char *topic,
                                    uint32_t *sec,
                                    uint32_t *nanosec,
                                    char *frame_id,
                                    size_t *frame_id_size,
                                    char *child_frame_id,
                                    size_t *child_frame_id_size,
                                    const bool no_covariance,
                                    double *x,
                                    double *y,
                                    double *z,
                                    double *quat_x,
                                    double *quat_y,
                                    double *quat_z,
                                    double *quat_w,
                                    double *covariance,
                                    double *vel_x,
                                    double *vel_y,
                                    double *vel_z,
                                    double *rot_x,
                                    double *rot_y,
                                    double *rot_z,
                                    double *vel_covariance);



    // -----  Publisher  -----

    /*
    topic: topic name
    msg: string to be published
    ret: true: success, false: fail                    */
    ROS_C_API bool send_string( const char *topic,
                                const char *msg);
```

```
/*
    topic: topic name
    double x, y, z: linear vector
    double rx, ry, rz: angular vector
    ret: true: success, false: fail                    */
    ROS_C_API bool send_twist(  const char *topic,
                                const double x,
                                const double y,
                                const double z,
                                const double rx,
                                const double ry,
                                const double rz);

    /*
    topic: topic name
    unsigned int sec, nanosec: timestamp
    frame_id: string
    ret: true: success, false: fail                    */
    ROS_C_API bool send_header( const char *topic,
                                uint32_t sec,
                                uint32_t nanosec,
                                const char *frame_id);

    /*
    topic: topic name
    double x, y, z: linear vector
    double rx, ry, rz: angular vector
    ret: true: success, false: fail                    */
    ROS_C_API bool send_pose(   const char *topic,
                                uint32_t sec,
                                uint32_t nanosec,
                                const char *frame_id,
                                const double x,
                                const double y,
                                const double z,
                                const double quat_x,
                                const double quat_y,
                                const double quat_z,
                                const double quat_w);
```

```
/*
    topic: topic name
    double x, y, z: linear vector
    double rx, ry, rz: angular vector
    cov_exists: bool
    covariance: float[36], or empty array if no covariance is given
            -> a matrix of zeros is put to the ROS2 message
    ret: true: success, false: fail                    */
    ROS_C_API bool send_pose_with_cov(  const char *topic,
                                        uint32_t sec,
                                        uint32_t nanosec,
                                        const char *frame_id,
                                        const double x,
                                        const double y,
                                        const double z,
                                        const double quat_x,
                                        const double quat_y,
                                        const double quat_z,
                                        const double quat_w,
                                        const bool cov_exists,
                                        const double *covariance);
}

#endif
```

APPENDIX C: CREATION OF A NEW PUBLISHER MESSAGE TYPE

Documentation was created to VTT intranet on how to add a new publisher message type:

1. add the ros package name to package.xml. For example:

   <depend>std_msgs</depend>

2. add the ros package to a CMakeLists.txt file of your project. For example:

   find_package(std_msgs REQUIRED)

   and

   ament_target_dependencies(your_target std_msgs)

   where **your_target** is your target library or executable project

3. add your package header files in your header files. For example:

   #include "std_msgs/msg/string.hpp"

4. UnityPublisher.h, under UnityROSPublisher::ROS_add_pub add a section for your message as such:

```
if(typeid(T) == typeid(std_msgs::msg::String))
  {
    if(publ_string_.find(topic) == publ_string_.end())
    {
      publ_string_.emplace(make_pair(
            topic, this->create_publisher<std_msgs::msg::String>(
            topic, backlog)));
      return true;
    } else
    {
      #ifdef DEBUG_MSGS
      printf("ERROR: A publisher already exists with the given topic\n");
      #endif
      return false;
    }
  }
```

5. UnityPublisher.h under UnityROSPublisher class add a dictionary(=map) for your publisher objects as such:

   unordered_map<std::string, PUBL_STRING_PTR> publ_string_;

6. UnityPublisher.h and UnityPublisher.cpp add ROS_send_pub method for the UnityROSPublisher class with your message type as parameter, for example

```
bool UnityROSPublisher::ROS_send_pub(const string& topic,
std_msgs::msg::String& msg)
{
   auto iter = publ_string_.find(string(topic));

   if(iter == publ_string_.end())
   {
      #ifdef DEBUG_MSGS
      printf("Error: no publishers with that topic name\n");
      #endif

      return false;
   } else
   {
      iter->second->publish(msg);
      return true;
   }
}
```

7. general_functions.h and general_functions.cpp: add your message type to add_publisher, as such:

```
      switch (type)
      {
      case 0:
         return pubObj->ROS_add_pub<std_msgs::msg::String>(       string(topic),
                                                                  ROS_backlog);
```

Note: Mind you take a unique type number for your message. See Publisher types for reserved type numbers.

8. overloader.h and overloader.cpp: add a C-compatible interface for the class function to send message, as such:

```
extern "C" {

   bool send_string(  const char *topic, const char *msg)
   {
      if(!ov_init)
      {
         #ifdef DEBUG_MSGS
         printf("Publisher not initialized\n");
         #endif
         return false;
      }


      #ifdef DEBUG_MSGS
      printf("SEND_PUB\n\ttopic: %s\n\ttype: string\n\tmsg: %s\n", topic, msg);
      send_debug("SEND_STRING: topic: " + string(topic));
      #endif

      const string topic_str = string(topic);
      const string msg_str = string(msg);

      std_msgs::msg::String ros_msg = std_msgs::msg::String();
```

```
        ros_msg.data = msg_str;

        return ov_pubObj->ROS_send_pub(topic_str, ros_msg);
    }
}
```

9. Document your changes to Confluence

APPENDIX D: CREATION OF A NEW SUBSCRIBER MESSAGE TYPE

Documentation was created to VTT intranet on how to add a new subscriber message type:

1. add the ros package name to package.xml. For example:

   <depend>std_msgs</depend>

2. add the ros package to a CMakeLists.txt file of your project. For example:

   find_package(std_msgs REQUIRED)
   and
   ament_target_dependencies(your_target std_msgs)

   where **your_target** is your target library or executable project

3. add your package header files in your header files. For example:

   #include "std_msgs/msg/string.hpp"

4. UnitySubscriber.h and UnitySubscriber.cpp add queue (=deque) for messages and a pointer for the subscription objects under TopicCallback class as such:

   ```
   deque<std_msgs::msg::String> msgs_string_;
   rclcpp::Subscription<std_msgs::msg::String>::SharedPtr subs_string_;
   ```

5. UnitySubscriber.h and UnitySubscriber.cpp add callback function under TopicCallback class as such:

   ```
   void TopicCallback::callback_string(
         std_msgs::msg::String::SharedPtr msg)
   {
       if(msgs_string_.size() >= MAX_MSG_BUFFER_SIZE_)
       {
           msgs_string_.pop_front();
       }
       msgs_string_.push_back(*msg);
       #ifdef DEBUG_MSGS
       printf("*** Callback: Got msg (topic: %s): %s ***\n",
                    topic_.c_str(), msg->data.c_str());
       #endif
   }
   ```

6. UnitySubscriber.cpp: Modify TopicCallback constructor to minimize memory usage, as such:

   msgs_string_.shrink_to_fit();

7. UnitySubscriber.h, under UnityROSSubscriber::ROS_add_sub add a section for your message as such:

```
if(typeid(T) == typeid(std_msgs::msg::String))
    {
        cout << "*** STRING ***" << endl;
        if(callbacks_.find(topic) == callbacks_.end())
        {
            empty = false;
            auto ret = callbacks_.emplace(make_pair(
                  topic,
                  TopicCallback(topic, DLL_backlog)));

            ret.first->second.subs_string_ = this->
                  create_subscription<T>(
                        topic,
                        ROS_backlog,
                        bind( &TopicCallback::callback_string,
                              &ret.first->second, _1));
            return true;
        }
    }
```

8. general_functions.h and general_functions.cpp: add your message type to add_subscriber, as such:

```
 switch (type)
 {
 case 0:
     return subObj->ROS_add_sub<std_msgs::msg::String>(
           string(topic),
           DLL_backlog,
           ROS_backlog);
```

Note: Mind you take a unique type number for your message. See Subscription types for reserved type numbers.

9. overloader.h and overloader.cpp: add a C-compatible interface for the class function to poll for a message, as such:

```
extern "C" {

    void poll_string(char* topic, char *out_buf, size_t *buf_size)
    {
        auto iter = ov_subObj->callbacks_.find(topic);
        auto iter_end = ov_subObj->callbacks_.end();
        std_msgs::msg::String temp;
        string msg;


        if(iter != iter_end)
        {
            rclcpp::spin_some(ov_subObj);

            if(!iter->second.msgs_string_.empty())
            {
                temp = iter->second.msgs_string_.front();
```

```
                    iter->second.msgs_string_.pop_front();
                    msg = temp.data;

                    if(msg.empty())
                    {
                        *buf_size = 0;
                        return;
                    } else if(msg.size() < *buf_size)
                    {
                        strncpy(out_buf, msg.c_str(), msg.size());
                        out_buf[msg.size()] = '\0';
                        *buf_size = msg.size();
                         return;
                    } else
                    {
                        strncpy(out_buf, msg.c_str() , *buf_size - 1);
                        out_buf[*buf_size - 1] = '\0';
                        *buf_size = *buf_size - 1;
                        return;
                    }
                }
            }

            // if no topic or msg is found
            *buf_size = 0;
            return;
        }
    }
```

10. Document your changes to Confluence

APPENDIX E: ROS1_TEST RESULTS

a) Time measurements of messages through ROS1_bridge

	ROS1 -> ROS2		ROS2 -> ROS1	
	Publisher first (s)	**Subscriber first (s)**	**Publisher first (s)**	**Subscriber first (s)**
	6,02	2,00	2,63	27,99
	6,29	2,05	1,46	32,33
	6,47	1,77	1,47	25,50
	5,27	1,71	2,11	14,80
	5,86	1,96	1,86	37,01
	5,94	1,80	1,72	34,45
	5,80	1,48	1,70	37,14
	5,92	1,58	2,30	36,01
	6,64	1,94	1,46	36,30
	6,57	2,19	1,51	36,82
Average:	**6,08**	**1,85**	**1,82**	**31,84**

b) Missed messages during time measurements, depicted in the previous table

	ROS1 -> ROS2		ROS2 -> ROS1	
	Publisher first	**Subscriber first**	**Publisher first**	**Subscriber first**
				7
				10
				3
				1
				15
				13
				15
				14
				14
				15
Average:				**10,7**

c) Time measurements of messages from ROS2 to ROS2 and ROS1 to ROS1

	ROS2 -> ROS2		ROS1 -> ROS1	
	Publisher first (s)	**Subscriber first (s)**	**Publisher first (s)**	**Subscriber first (s)**
	2,25	6,06	2,45	1,15
	3,14	5,98	2,34	1,07
	3,18	5,99	2,36	1,09
	4,41	6,00	2,40	1,13
	4,15	5,97	2,33	1,10
	3,97	6,00	2,35	1,15
	4,12	6,01	2,34	1,07
	4,48	6,00	2,33	1,05
	2,11	5,98	2,47	1,10
	4,38	6,08	2,28	1,06
Average:	**3,62**	**6,01**	**2,37**	**1,10**