INDUSTRIAL PROCESSES IN AUTOMOTIVE ENGINEERING

Academic Tutor: prof. CHIABERT PAOLO

Master Thesis

MIND LAB DIGITAL TWIN COMMUNICATION ARCHITECTURE USING NODERED AND MQTT

Automotive engineering

Student:

Sai Nagarjun Bandaru – S301794

Declaration

I hereby declare that except where specific reference is made to the work of others, the contents of this dissertation are original and have not been submitted in whole or in part for consideration for any other degree or qualification in this, or any other university. This dissertation is my work and contains nothing which is the outcome of work done in collaboration with others, except as specified in the text and Acknowledgements. This dissertation contains fewer than 6450 words including appendices, bibliography, footnotes, tables, and equations, and has fewer than 41 figures.

Bandaru Sai Nagarjun

March 2024

Acknowledgments

I am sincerely grateful to Prof. CHIABERT PAOLO of the POLITECNICO DI TORINO for his invaluable guidance and unwavering support throughout my thesis work.

I am sincerely grateful to MANSUR ASRANOV. From the beginning, MANSUR ASRANOV demonstrated trust in my abilities and constantly encouraged me to excel. His expertise and mentorship as my supervisor were truly invaluable.

I extend my heartfelt thanks to MAHKAM KAYUMOV for his technical expertise and instrumental support in the success of my thesis work. His insightful suggestions greatly enhanced my research and professional growth.

I sincerely appreciate my family's unwavering support and belief in my capabilities throughout my academic journey, which has constantly motivated me. I am also grateful for the enriching and memorable international student life in Italy, thanks to the friendship and companionship of my friends.

My deepest gratitude goes to Prof. CHIABERT PAOLO, Mansur, my family, friends, and all those who supported me in my thesis work and career development. Their unwavering support and encouragement have been vital to my success.

Table of Contents

Abstract

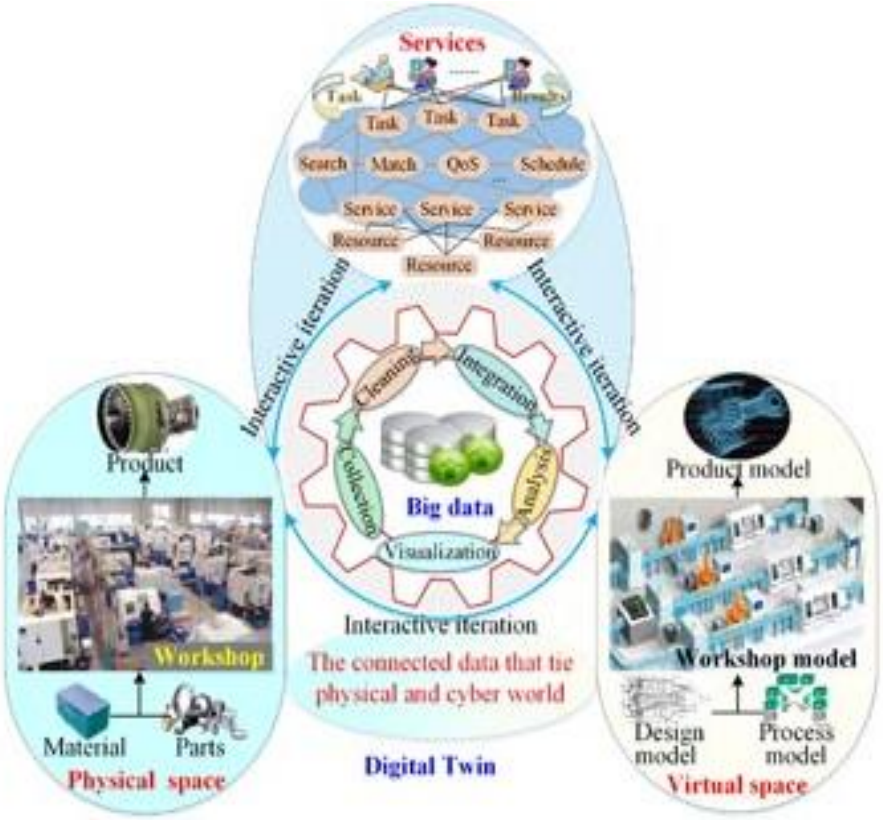
The main goal of this thesis is to design an Advanced Industrial 4.0 oriented Digital Twin for Simulation of different robots in the Mind-Lab of Politecnico Di Torino, where all the Robots Uses different communication protocol. MQTT Brocker is used to collect the Data from all the robots as universal platform. This possible communication architecture is more reliable, secured, and flexible opening new gates for further adding new robots and sensors in the lab and lets controllability.

1. Introduction

Industrial 4.0 Digital Twin, the idea of a "digital twin" has become more popular as industrial processes become more digital. A digital twin is an almost real-time virtual copy of a real-world item or procedure that is used to maximize corporate efficiency. Digital twins function as virtual copies of factories, supply chains, production lines, and processes in the context of Industry 4.0. Data from IoT sensors, devices, PLCs, and other linked items are pulled to generate these twins. Digital twins let companies spot physical problems early, anticipate outcomes accurately, and create better products by giving them access to a product's whole digital footprint. Digital twins can help businesses increase their speed to market, improve their operations, lower error rates, and investigate new business opportunities. For businesses looking to revolutionize their production processes, the digital twin is increasingly accessible as costs come down and technological capabilities advance.

Manufacturing: Businesses use Internet of Things sensors to build digital twins of actual factories. Predictive maintenance, quality control, and production process monitoring are made possible by these twins. Manufacturers can increase productivity, decrease downtime, and improve efficiency by simulating various scenarios.

Automotive: Vehicle testing, design, and performance optimization are all done with digital twins. Digital twins are useful for large engines (like locomotive and jet engines), particularly for creating maintenance schedules.



1.1. Origin of Digital Twin

The idea of digital twins (DT) began with the early stages of virtual manufacturing (VM), which Onosato and Iwata presented in 1993. VM used modelling and simulation environments to generate virtual representations of physical systems. What is today known as Digital Twins originated from this fundamental concept of virtual machines. In contrast to what some people say, DT came from the idea of VM rather than the mirrored space model (MSM) put forth by Michael Grieves. Although it lacked a clear meaning, the term "DT" first surfaced in literature in 1997 when it was used to describe a three-dimensional digital model of metropolitan road networks. With its development based on the integration of interdisciplinary technical knowledge, industrial software, and information and communication technologies (ICT), DT has attracted a great deal of interest over time from both industry and academia. The development and useful uses of digital twins in computational engineering and sophisticated system modelling and simulation settings are highlighted by this historical background.

The idea of "digital twins" grew as technology progressed, moving from basic virtual representations to more intricate, integrated systems with real-time data processing and sophisticated analytics. Advances in IoT, cloud computing, big data, and AI facilitated this transition by giving the tools and infrastructure needed to construct dynamic, highly detailed digital twins. Michael Grieves first explicitly introduced the concept of a digital twin in relation to product lifecycle management (PLM) in a 2002 presentation at the University of Michigan. His idea, which centred on the synchronization of the actual and virtual worlds, served as the basis for our modern understanding of digital twins. Grieves' three main components were the physical product, the virtual product, and the connections that brought the two together through data and information flows.

Many various industries, including smart cities, healthcare, automotive, and energy, are currently using digital twin technologies. In the medical field, digital twins of patients offer more customized therapy and improved treatment results. In the automotive industry, they assist with vehicle design, testing, and performance improvement. In the energy sector, digital twins maximize the operation and maintenance of power plants and renewable energy systems. Smart cities employ digital twins for urban planning and infrastructure management. The further advancement and integration of cutting-edge technologies like artificial intelligence (AI), machine learning, and edge computing are expected to greatly expand the capabilities and uses of digital twins. As digital twin technology develops, it has the potential to revolutionize whole industries by boosting system intelligence, product reliability, and operational productivity. The evolution of digital twins over time is proof of their revolutionary potential as well as the ongoing innovation that drives their development.

1.2. Digital Twin in Industrial 5.0

The development of Industry 4.0, a technological paradigm shift that incorporates digital technology into industrial and manufacturing processes, is closely linked to the birth of the notion of the digital twin. The fourth industrial revolution, known as "industry4.0," is defined by the fusion of cutting-edge digital and physical technologies, such as Big data analytics, Autonomous Robots, System integration, Cybersecurity, Additive manufacturing,

Augmented reality, artificial intelligence, and the internet of things (IoT). The ground-breaking idea of digital twins (DT) was first proposed within the context of Industry 4.0. Later in January 2021 the European commission categorized DT as Industrial 5.0 technology towards a human centric, sustainable and resilient European industry. Through ongoing data processing and analysis, it aims to optimize production processes and facilitate proactive maintenance. The idea behind digital technology (DT) is to create a "digital copy" or virtual representation of real production systems, including assembly lines or other pieces of machinery. With the help of this digital copy, businesses can monitor, simulate, and optimize operations in real time, enabling them to quickly modify production parameters and procedures in response to market needs.

"Digital Twin" refers to the merging of digital and physical worlds in a digital manufacturing setting. Using real-time data gathered from physical assets, a DT gives businesses deep insights into the effectiveness of their products and production processes. This data-driven approach to decision-making helps identifies inefficiencies, shorten manufacturing cycles, and speed up the introduction of new products. The Digital Twin concept, which leverages networked technology to increase manufacturing efficiency, agility, and competitiveness, is the fundamental embodiment of Industry 4.0. As a result, DTs are now a crucial part of modern industrial operations, supporting the transition to data-driven, intelligent manufacturing systems that are adaptable, efficient, and responsive to shifting market conditions.

Digital twins have a wide range of effects in Industry 5.0, going beyond simple operational gains to include more significant strategic advantages. Manufacturers can anticipate probable equipment breakdowns and schedule maintenance to minimize downtime and prolong the life of machinery, for instance, by integrating DTs with predictive analytics and machine learning. The capacity to do predictive maintenance is essential for preserving continuous production flows and lowering expenses related to unplanned equipment failures. Additionally, digital twins enable better teamwork at different phases of the product lifecycle. DTs facilitate speedier prototyping, more efficient design iterations, and more efficient product development processes by offering a full virtual model that many teams can view and edit. This collaborative setting guarantees that all parties involved—from engineers to designers to maintenance teams—have a single, accurate understanding of the product and its state of operation, which promotes more aligned goals and superior results.

Digital twins provide real-time monitoring and control over inventories and logistics in the context of supply chain management. Businesses may enhance delivery timetables, optimize inventory levels, and foresee disruptions by modelling various supply chain scenarios. This degree of understanding is especially helpful in the increasingly complicated and multinational supply chain environment, where preserving competitive advantage depends on responsiveness and agility. Moreover, the use of digital twins is consistent with industry 5.0's rising emphasis on sustainability and human centric application. With their ability to maximize resource use, minimize waste, and enhance energy efficiency, DTs help to promote more environmentally friendly industrial methods. Companies that integrate their operations with regulatory standards and corporate sustainability goals can monitor and manage their environmental impact more efficiently.

To sum up, the incorporation of Digital Twins into the Industry 5.0 paradigm signifies a noteworthy progression in the way industrial processes are carried out. Through the integration of the digital and physical realms, DTs offer a strong foundation for creativity, effectiveness, and strategic expansion. The future of manufacturing will be shaped by Digital Twins, who will play an increasingly significant role as technology advances and push the sector toward more intelligent, robust, and sustainable operations.

1.3. Digital Twin in Production line

In today's manufacturing landscape, the utilization of a digital twin within production lines extends beyond mere virtual representation. Manufacturers may use advanced analytics and simulations to make pre-emptive decisions thanks to this cutting-edge technology. A digital twin records real-time data from sensors included into the production environment by constantly syncing with its physical counterpart. This wealth of data enables predictive maintenance strategies, as anomalies and potential issues can be detected and addressed before they impact operations. Within the framework of body-in-white (BIW) manufacturing systems, a digital twin provides crucial understanding of the intricate interactions among machinery, procedures, and final goods. It can, for instance, detect production flow bottlenecks, keep an eye on the well-being and efficiency of robots, and adjust resource usage to suit changing demand. Using a digital twin also makes it easier for engineers, maintenance personnel, and production planners to communicate with one another and encourages collaboration among interdisciplinary teams.

As companies adopt Industry 5.0 concepts, the use of digital twins in production has the potential to revolutionize operational responsiveness and efficiency. Manufacturers may use data-driven decision-making and predictive analytics to improve the agility, quality, and efficiency of their manufacturing processes. The digital twin, a crucial element in the shift to smarter, more flexible production settings, is ushering in an era of greater innovation and competition. Digital technologies like AI and Robotics can reshape and optimize human-machine interactions mainly in the production line with human workers on factory floors.

1.4. Digital Twin in automotive

The incorporation of Digital Twin technology is causing a disruptive shift in the automotive industry by improving vehicle performance, design, and maintenance. Digital twins can enhance designs, facilitate predictive maintenance, and boost safety using advanced driver assistance systems (ADAS) by replicating real-time vehicle data. From early CAD design and quick prototyping to post-market performance analysis and personalized customer experiences, this technology serves the whole vehicle lifecycle. As demonstrated by Tesla's use of this technology, manufacturers can utilize Digital Twins to model the behaviour of auto parts, predict failure rates, and apply over-the-air software upgrades to solve problems in real time. Additionally, by enabling prospective purchasers to digitally alter the appearance of vehicles, digital twins enable unique consumer experiences. The increasing developments in IoT and AI suggest that Digital Twins will be essential to creating smarter, more efficient, and sustainable vehicles, which will have a substantial impact on the future of the automotive industry, despite obstacles in data management, networking, and system integration.

1.5. Multi Robot integration in Digital twin using MQTT

When combined with MQTT brokers, digital twins provide reliable and effective communication frameworks for the interchange of data in real time among several robots in complex systems found in smart manufacturing settings. Due to its lightweight design and dependability, the MQTT protocol makes it easier to communicate with physical entities and the Digital Twin by guaranteeing that messages are sent exactly once, which is a crucial aspect of data integrity management in dynamic production environments. Different parts of the Digital Twin ecosystem can publish to and subscribe to topics using a locally hosted, open-source MQTT broker such as Mosquitto, allowing for real-time updates and synchronization. With this configuration, the Digital Twin can accurately represent the current conditions of the robots, improving the capacity for predictive maintenance and decision-making. Additionally, MQTT is a great option for Internet of Things applications due to its scalable architecture and low power consumption, which helps smart manufacturing systems operate sustainably and effectively. Digital Twins can dependably handle data from several sources using MQTT, guaranteeing that the system functions as a whole and reacts quickly to any demands or changes in operation.

2. Problem Description

A Cyber-Physical System (CPS) with Robots, Cobots, and Mobots will be created as a Digital Twin model in this thesis. In order to encourage wider adoption and cooperation, the CPS will primarily make use of open-source tools, and the completed environment will be made available on open-source platforms. The environment will be user-friendly and able to support extensive simulation and interaction thanks to the CPS's processes, devices, and interfaces for signal monitoring and regulation. Integrating the digital twins of the three distinct robots housed in the Politecnico di Torino Mind Lab is the main goal of the thesis. Among these robots are the 60 kg payload carrying ABB IRB4400, the UR3, and the UR2 installed on a mobile robot (AGV). Real-Time Data Exchange (RTDE) is used by the UR3 and UR2 robots, the MODbus communication protocol is used by the mobile robot, and the API communication protocol is used by the ABB robot. Developing a dependable and coherent communication architecture for the CPS is extremely difficult because to the various features, communication protocols, and mechanisms that each robot possesses.

Furthermore, for the CPS to operate efficiently, smooth interoperability between these disparate systems is essential. It is necessary to carefully include each robot's distinct communication protocol into a single framework that enables real-time data sharing and coordination. Achieving coordinated operations and maximizing the CPS's overall performance depend on this integration. Creating an effective plan for data management is another major difficulty. Robust data handling and processing procedures are required for the CPS, as each communication protocol has distinct data formats and transmission frequencies. This involves offering scalable data storage solutions, reducing latency, and guaranteeing data integrity. In order to provide operational optimization, performance monitoring, and predictive maintenance, the CPS must also support real-time analytics. To allow for future additions and updates, the CPS also needs to be designed with scalability and flexibility in mind. This covers the possible introduction of fresh sensors, robotics, and other Internet of thing’s gadgets. Because the project is open source, more researchers and developers will be

able to contribute to its ongoing customization and enhancement, which will promote creativity and adaptability.

The thesis will also investigate how incorporating cutting-edge technology like machine learning and artificial intelligence might enhance the capabilities of the Digital Twin model. These technologies make it feasible to assess the massive amounts of data generated by the CPS and obtain knowledge that may improve decision-making, predictive maintenance, and operational efficiency. Finally, operators will be able to communicate with the CPS from any location because to its capabilities for remote monitoring and control. With the advent of Industry 5.0 and the growing prevalence of distant and decentralized operations, this capability is especially pertinent. A key component of the system's architecture will be to ensure dependable and secure remote access, which calls for the deployment of strong cybersecurity safeguards.

In conclusion, creating a CPS that combines Digital Twins of various robot types with various communication protocols is a difficult but incredibly satisfying task. The thesis seeks to develop a flexible and scalable environment that not only satisfies present operating requirements but also lays the groundwork for upcoming developments in cyber-physical systems and Industry 5.0 applications by utilizing open-source tools and platforms.

In the thesis, a Cyber Physical System (CPS) comprising Robots, Cobot’s, and Mobot’s is developed as a Digital Twin model. The CPS will be developed mostly using open-source tools, and the finished environment will be made accessible on open-source platforms. It will be simple to interact with the environment and simulate it thanks to the CPS's mechanisms, gadgets, and interfaces for monitoring and regulating signals. The main aim of the thesis is to integrate Digital Twins of 3 different robot existing in the Politecnico Di Torino Mind Lab. The robots include ABB IRB4400 payload 60kgs, UR3 and UR2 mounted on mobile robot (AGV). The ABB robot uses API communication protocol, UR3 and UR2 uses RTDE, and the mobile robot uses Modbus communication protocol. Developing a dependable communication architecture for the cyber-physical system is difficult since every robot has unique features, communication protocols, and mechanisms.

3. Description of Communication Protocols of Robots

3.1. ABB IRB 4400

ABB Robot

3.1.1. ABB Axis:

With six axes that are each intended to enable precise and adaptable movement for a variety of industrial applications, the ABB IRB 4400 is an incredibly sturdy and versatile industrial robot. The axes are:

1. **Axis 1 (Base Rotation)**: This axis gives the robot a full 360-degree swivel capability by enabling it to revolve around its vertical base. For tasks requiring a wide range of placement and movements, this is essential.
2. **Axis 2 (Arm Extension)**: The robot can grasp items at varying distances thanks to this axis, which allows the arm to extend forward and backward.
3. **Axis 3 (Arm Elevation)**: The arm may move up and down on this axis, making it easier to handle things of varying heights.
4. **Axis 4 (Wrist Rotation)**: The wrist may rotate on this axis, which improves the robot's capacity to carry out intricate tasks including twisting motions.
5. **Axis 5 (Wrist Bend)**: The wrist may be bent thanks to this axis, which further improves the robot's dexterity and adaptability.
6. **Axis 6 (End Effector Rotation)**: The end effector, or tool attached to the robot's arm, can spin precisely on this axis, facilitating sophisticated and detailed operations.

These six axes work together to provide the IRB 4400 remarkable flexibility and precision, making it perfect for demanding tasks in a variety of industrial applications.

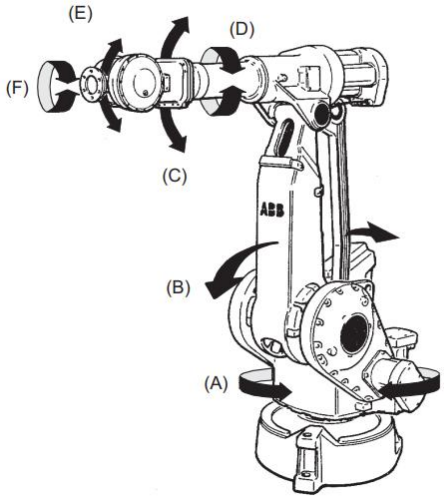


ABB AXIS

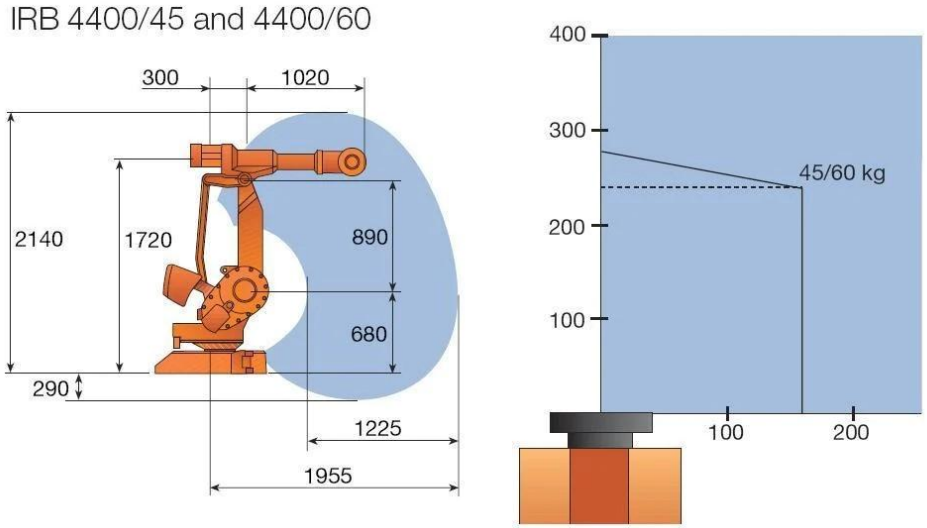
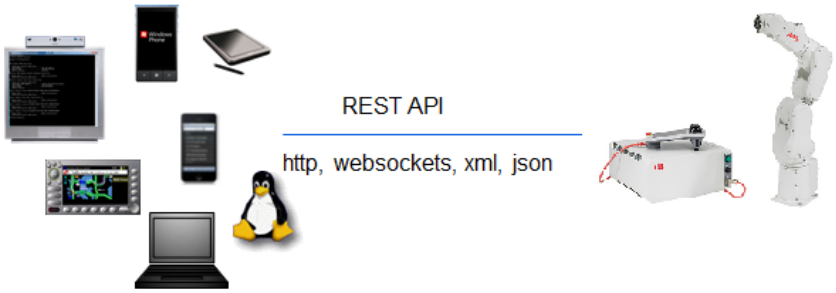


ABB REACHABLE WORKSPACE

3.1.2. Robot available Communication Protocols

A variety of communication protocols are used by the ABB IRB 4400 robot to guarantee effective and seamless integration into automated systems. Important protocols consist of:

- **API Communication Protocol:** The robot's controller and other devices or systems can communicate with each other more easily thanks to the IRB 4400's use of API (Application Programming Interface) protocols. Real-time data sharing and command execution are made possible by this protocol, which is crucial for coordinated operations in automated settings.



API Network

- **Modbus Communication Protocol:** The Modbus protocol, a widely used standard in industrial automation, is another protocol that the IRB 4400 can employ to communicate between electrical devices. By facilitating communication with other robots, HMIs (Human-Machine Interfaces), and PLCs (Programmable Logic Controllers), Modbus enhances interoperability within a manufacturing line.

- **Robot Ware and IRC5 Controller:** A key component of the IRB 4400's communication capabilities is its Robot Ware-equipped IRC5 controller. With support for many connectivity methods like Ethernet, Device Net, and PROFIBUS, this combination ensures dependable and strong data transmission with external devices and systems.

These communication protocols enable coordinated operations and real-time data transmission, and they also make it simple to integrate the IRB 4400 into complex automated systems.

3.1.3. Robot Uses and Functioning

The ABB IRB 4400's sophisticated features and sturdy construction make it ideal for a variety of industrial applications. Important applications consist of:

- **Manufacturing:** The IRB 4400 is widely utilized in the manufacturing sector for operations like machine tending, assembly, material handling, and packaging. Because of its great speed and accuracy, it is ideal for repetitive tasks requiring accuracy and efficiency.
- **Welding and Gluing:** The IRB 4400 is capable of welding and gluing, providing exact control over the application of welds and adhesives, with the addition of optional software packages.
- **Material Removal:** The robot can do tasks that call for steady force and accuracy, like cutting, grinding, and polishing.

- **Die-Casting and Foundry Applications:** The IRB 4400 is appropriate for die-casting and other foundry applications because of its Foundry Plus feature, which enables it to function in challenging conditions and withstand exposure to coolants, lubricants, and metal spits.
- **Quality Inspection:** To ensure that items satisfy strict quality requirements before they are released onto the market, quality inspection jobs can be performed by a robot that is outfitted with vision systems and sensors.

The IRB 4400 boosts productivity, decreases downtime, and increases overall efficiency in industrial processes by combining these uses and features. Because of its dependability and versatility in managing a variety of tasks, it is an essential tool in modern automated manufacturing systems.

3.2. API

3.2.1. API Definition and Role

An Application Programming Interface (API) is a collection of guidelines and conventions that facilitates communication between various software programs. According to this thesis, the API is essential to allowing smooth data transfer via Node-RED between the ABB IRB 4400 robot and the MQTT broker. The ABB robot collects operational data in real time, including joint locations, movement trajectories, and performance parameters, by using its API. After that, this data is sent to the flow-based programming tool Node-RED, which prepares and processes it before sending it to the MQTT broker. Real-time monitoring, simulation, and optimization of the robot's actions within the digital twin environment are made possible by the system's utilisation of the API to guarantee that the digital twin receives precise and timely data. This integration not only enhances the interoperability of the different components within the Cyber-Physical System but also ensures that the digital twin can effectively mirror the physical operations of the ABB robot, thereby contributing to the overall efficiency and reliability of the manufacturing processes.

3.2.2. API for Robot Communication

APIs are essential to contemporary robotics because they offer a standardized means of communication and interaction between various components and systems. Robots can send and receive data, carry out commands, and easily connect with other systems and applications thanks to APIs in robot communication. The API makes it easier for other systems to communicate with the control software of the ABB IRB 4400 robot by making a variety of endpoints available. Real-time status monitoring, command transmission for movement, and sensor data retrieval are all part of this interaction.

The API plays a vital role in this thesis by acting as a link between the ABB robot and the larger digital twin environment. The robot can upload its data to the MQTT broker using Node-RED by using the API. In this procedure, joint data, location data, and operating statuses are transmitted by the robot's API to Node-RED, which processes and passes the data to the MQTT broker. As a message mediator, the MQTT broker makes sure that all subscribing clients—including the Unity-developed digital twin system—get access to this

data. This configuration improves the monitoring and control capabilities within the lab's smart manufacturing environment by enabling real-time visibility and interaction with the robot's digital twin.

3.2.3. ABB ROBOT API

ABB robot integration into contemporary industrial environments, particularly in the context of Industry 4.0 and the creation of Digital Twins, requires the use of the ABB API Communication Protocol. ABB robots and external systems like control software, monitoring apps, and digital interfaces may communicate data and commands with ease thanks to this protocol's reliable and effective bidirectional communication. A broad range of features intended to meet various operational requirements form the foundation of the ABB API. These consist of the real-time data transfer of operating statuses, error messages, joint positions, and other crucial characteristics. The efficient execution of automated operations, problem diagnostics, and robot performance monitoring all depend on this kind of rapid and comprehensive information.

3.2.4. Technical overview

The ABB API is compatible with a variety of network setups and client applications since it uses common communication protocols like HTTP and WebSockets. The API endpoints are made to perform a wide range of tasks, from simple command execution to sophisticated data extraction. Token-based authentication is one of the industry-standard authentication techniques used to secure each endpoint, guaranteeing that only authorized users and systems can communicate with the robot controllers.

The ABB API's capability to handle real-time data streams is one of its primary characteristics. This is especially significant when implementing a digital twin, as accurate modelling and monitoring depend on current data. The API enables the flow-based programming tool Node-RED, which is a versatile and user-friendly tool, to upload data in real-time from ABB robots to a MQTT broker. By serving as a middleman, Node-RED gathers information from the ABB robot controls and sends it to the MQTT broker. By ensuring that the Digital Twin receives up-to-date and correct information, this configuration improves its ability to simulate, monitor, and optimize robotic activities.

3.2.5. API Security and Reliability

For the ABB IRB 4400 robot and other systems to communicate safely and consistently, it is imperative that the API's security and dependability be guaranteed. Token-based authentication, HTTPS, and encryption are examples of security mechanisms that guard against unwanted access and data breaches. Robust error-handling, redundancy, and failover solutions are employed to preserve reliability, guaranteeing continuous real-time synchronization between the physical robot and its Unity digital twin. The digital twin's sophisticated features are supported by this reliable and secure communication framework, which also maintains data integrity and system resilience.

3.3. MQTT Broker

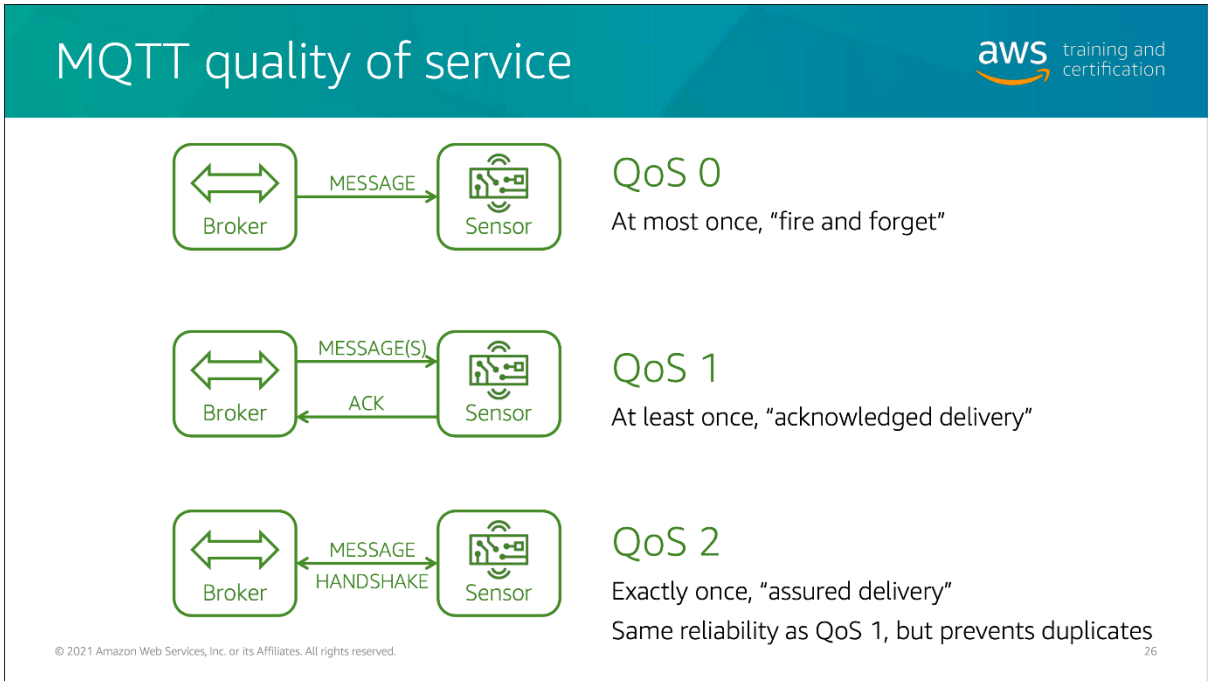
3.3.1. Overview of MQTT

A key component of the Digital Twin system's communication architecture is the MQTT Broker. It serves as a go-between, allowing the various robots and the Digital Twin environment to communicate with each other. Because of its dependability and lightweight design—which includes a publish-subscribe mechanism that effectively manages data transfer with little bandwidth and power consumption—MQTT is favoured. This is especially crucial in industrial settings where a multitude of equipment need to interact with one another without any problems. The broker supports Quality of Service (QoS) levels, ensuring message delivery with varying degrees of assurance, which is critical for maintaining the accuracy and reliability of the system. The MQTT Broker ensures seamless integration and real-time data synchronization, allowing for accurate representation and control of the ABB IRB 4400, UR3, and the mobile robot within the Unity-based virtual environment. This setup enhances the system's responsiveness and scalability, crucial for complex industrial applications. By leveraging MQTT, the system can adapt to changes and expansions, providing a robust framework for the Digital Twin's ongoing development and operational needs.

3.3.2. Advantages of Using MQTT

There are a number of clear benefits to integrating various robot types, such as ABB robots, UR3, and mobile robots, using MQTT as the main communication protocol. MQTT is perfect for areas with constrained network resources because of its lightweight design and minimal bandwidth requirements. This is especially useful in a laboratory environment where several robots are working in tandem and sending out data all the time. The publish-subscribe paradigm of MQTT facilitates effective data distribution, allowing robots and the digital twin system to communicate in real time. This paradigm also facilitates the integration of robots with different communication protocols, as MQTT may operate as a unifying layer that abstracts the complexities of individual protocols.

One of the key advantages of MQTT is its Quality of Service (QoS) levels, which ensure reliable message delivery. This function ensures that important data is not lost during transmission, which is essential for preserving the digital twin's correctness and synchronization. Depending on the demands of a particular application, the QoS levels vary from 0 (at most once delivery) to 2 (exactly once delivery), offering flexibility in balancing dependability and network resource utilization.



MQTT's robustness and dependability are further increased by its support for last will and testament features and preserved messages. Preserved messages guarantee that a fresh recipient gets the most recent message sent about a subject right away, which is crucial for setting up freshly linked robots' states. In the event that a robot unintentionally disconnects, the last will testament feature enables the system to alert other devices, guaranteeing timely and suitable responses to such occurrences.

Another important benefit of MQTT is its scalability. MQTT can accommodate the growing number of devices and message throughput as the lab's robotic fleet expands without requiring major infrastructure modifications. The effective message routing and topic-based filtering built into the MQTT protocol enable this scalability.

Finally, managing the communication flows is made simple and effective by using MQTT with Node-RED. The visual programming interface of Node-RED makes it simple to integrate, monitor, and operate the robots, facilitating rapid troubleshooting and adaption. This combination improves the lab's digital twin system's overall efficiency and versatility while streamlining the process of connecting various robots. The lab can create a scalable, dependable, and effective communication infrastructure that facilitates the smooth operation and coordination of its many robotic systems by utilizing MQTT.

3.3.3. MQTT Implementation in the Digital Twin

ABB IRB4400, UR3, and mobile robots are just a few of the different robots that our lab's robotic systems are controlling, synchronizing, and communicating with ease through the use of MQTT in the Digital Twin architecture. This requires a number of calculated steps. Because of its effectiveness, dependability, and scalability—all of which are essential for real-time data processing and system integration in a dynamic lab setting—the MQTT protocol was selected.

Establishing a reliable MQTT broker, like Mosquitto, to act as the focal point of all message exchanges is the first stage in the implementation process. The broker is in charge of overseeing the publish-subscribe process, making sure that all subscribing organizations receive data published by robots or control systems in a timely manner. The precision and efficacy of the Digital Twin depend on real-time monitoring and control, which is made possible by this configuration.

Every robot in the lab is set up to handle data transmission and reception using a MQTT client. For example, the ABB IRB4400 robot sends operational data to Node-RED via its API, and Node-RED broadcasts the data to pertinent MQTT topics. Similar to this, UR3 and mobile robots communicate with Node-RED using their own communication protocols (RTDE and MODbus). Node-RED serves as a middleman, translating and publishing data to the MQTT broker. This method makes sure that all robots may communicate with each other without any issues using a single MQTT-based framework, even though their native communication protocols differ.

The hierarchical nature of the MQTT topics makes data organization effective and access simple. Different data kinds, including position, status, and commands, have defined topics that enable subscribers to filter and analyse just the pertinent data. As more robots and gadgets are incorporated into the Digital Twin, this hierarchical subject structure also helps to manage the system's scalability.

In order to guarantee dependable message delivery, the system additionally incorporates techniques to manage quality of service (QoS) levels. QoS level 2 guarantees that every message is sent exactly once for critical data that needs to be delivered with certainty. For the physical robots and their digital counterparts to remain in sync and for the Digital Twin to correctly represent real-world conditions, this dependability is essential.

Retained messages are also utilized to keep each robot's most recent known state, guaranteeing that new subscribers can get the most recent status right away upon connecting. In order to detect and manage unexpected disconnections, MQTT's last will, and testament feature is also utilized. This helps the system remain strong and dependable even when there are network problems.

The SolidWorks models of the robots and the lab setup are animated using Unity in order to see and interact with the Digital Twin. The animations are powered by real-time data obtained from the Unity application, which subscribes to the pertinent MQTT topics. This allows for a dynamic and interactive representation of the lab's activities. To ensure that the digital models faithfully replicate the movements and actions of the real robots, Unity uses C# scripts to process joint and position data.

In conclusion, the integration of MQTT into the lab's robotic systems' Digital Twin framework offers a dependable, expandable, and effective communication option. It makes it possible for various robot kinds to synchronize and exchange data in real-time, which improves lab operation monitoring, control, and optimization. This integration opens the door for cutting-edge robotics and automation research and development in the laboratory setting in addition to increasing operational efficiency.

4. Design and Modelling of the Digital Twin

This part contains the comprehensive design and development of the lab layout and robotic systems SolidWorks models utilized in our Digital Twin architecture. The main goals are to produce precise and useful 3D models of the ABB IRB4400 robot and to integrate the mobile robot and UR3 models that already exist to enable efficient simulation and visualization.

Known for its great payload capacity and adaptability, the ABB IRB4400 robot is an essential part of our lab. Its movements and physical attributes were carefully replicated in the SolidWorks model through great craftsmanship. With all six of the robot's axes included, the model faithfully captures the kinematics and range of motion of the device. The joint configurations and link dimensions were carefully considered in order to guarantee that the virtual twin acts in the same way as the actual robot in real-world situations. We seek to attain great fidelity in our simulations by integrating the unique geometric and functional characteristics of the ABB IRB4400, which will increase the dependability of our digital twin.

Our digital architecture included pre-existing SolidWorks models of the UR3 and the mobile robot in addition to the ABB IRB4400. A collaborative robot with a reputation for accuracy and flexibility, the UR3, was included to mimic activities requiring a high level of precision and flexibility. To accurately depict its operational capabilities, the model incorporates end-effector combinations and intricate joint mechanisms.

The mobile robot was also modelled and integrated, and it had navigational skills within the lab. Its motion characteristics, sensor locations, and environmental interaction processes are all captured in this model. Our capacity to plan and optimize workflows is improved by the inclusion of the mobile robot in the digital twin, which enables thorough simulations of dynamic interactions inside the lab area.

The overall lab layout was modelled in SolidWorks to provide a comprehensive virtual environment. The lab layout includes workstations, robot operating areas, and pathways for the mobile robot. This holistic modelling approach enables a thorough understanding of spatial dynamics and interaction between different robotic systems within the lab. The detailed layout also assists in planning and optimizing workflows, ensuring efficient use of space and resources.

The finished SolidWorks models were saved as STL files and then imported into Blender. These models were fine-tuned in Blender and exported as Unity-compatible FBX files. This conversion procedure is essential because Unity needs FBX files in order to render and animate the simulation environment correctly. We make sure the models are intact and functional when we import them into Unity by using Blender as a transitional step.

Subsections that follow will include illustrations of these SolidWorks models, highlighting the fine details of the ABB IRB4400, UR3, and mobile robot in addition to the detailed lab architecture. These images show off SolidWorks' ability to produce accurate and useful digital twins for cutting-edge robotics applications in addition to illuminating the design process.

4.1. ABB IRB 4400 model

An essential component of our Digital Twin architecture is the industrial robot ABB IRB 4400, which is a strong and adaptable machine meant for high-precision work. This section explores the in-depth SolidWorks modelling of the ABB IRB 4400, highlighting the complex design process of its 13 mechanical components and how they are assembled to create a complete 6-axis robotic system. The digital model faithfully replicates its physical counterpart thanks to the careful crafting of each component to match the physical characteristics and movement capabilities of the real robot. Creating each component with exact measurements and guidelines, then assembling them to precisely mimic the robot's mechanical design and joint movements, was the modelling phase.

For our digital simulations, the SolidWorks models provide the framework. Based on the precise specifications of the real ABB IRB 4400 robot, each of the 13 parts—base, arm, and different joints—was developed separately. Since it makes accurate simulations and testing scenarios possible, this attention to detail is essential to the digital twin's correctness.

The SolidWorks files were saved as STL files, which were then imported into Blender in order to incorporate these models into the Unity simulation environment. The models were transformed into Unity-compatible FBX files using Blender. In order to preserve the models' functionality and integrity throughout animation and interaction within the digital twin, this conversion process is essential. High-quality visuals and interactions within Unity are made possible by the FBX format, which guarantees the preservation of intricate geometry and textures.

The subsequent pictures and explanations show the component pieces as well as the assembled product, emphasizing the accuracy and meticulousness required to produce a dependable digital twin for sophisticated simulation and operational planning. We can perform extensive testing, debugging, and optimization of robotic operations in a virtual environment by correctly modelling the ABB IRB 4400, which greatly expands the capabilities of our digital twin system.

Existing UR3 and mobile robot types were also used in addition to the ABB IRB 4400. These models underwent comparable processing, which included SolidWorks design verification and optimization before Blender converted the models into FBX files. All robotic elements within the digital twin are guaranteed to retain excellent quality and interoperability thanks to this standardized process.

In order to replicate cooperative activities with the ABB IRB 4400, the UR3 model—a smaller but no less accurate robotic arm—was incorporated into the configuration. In addition to being modelled and ready for Unity integration, the mobile robot—which is intended for independent navigation inside the lab setting—also allowed for dynamic simulations of movement and interaction with the stationary robotic arms.

When combined, these individual models produce a comprehensive and adaptable digital twin that can replicate a variety of interactions and situations in the laboratory. In addition to facilitating precise and effective design and testing, the procedure offers a solid foundation for upcoming additions and improvements to the digital twin environment.

ABB Solid Parts:

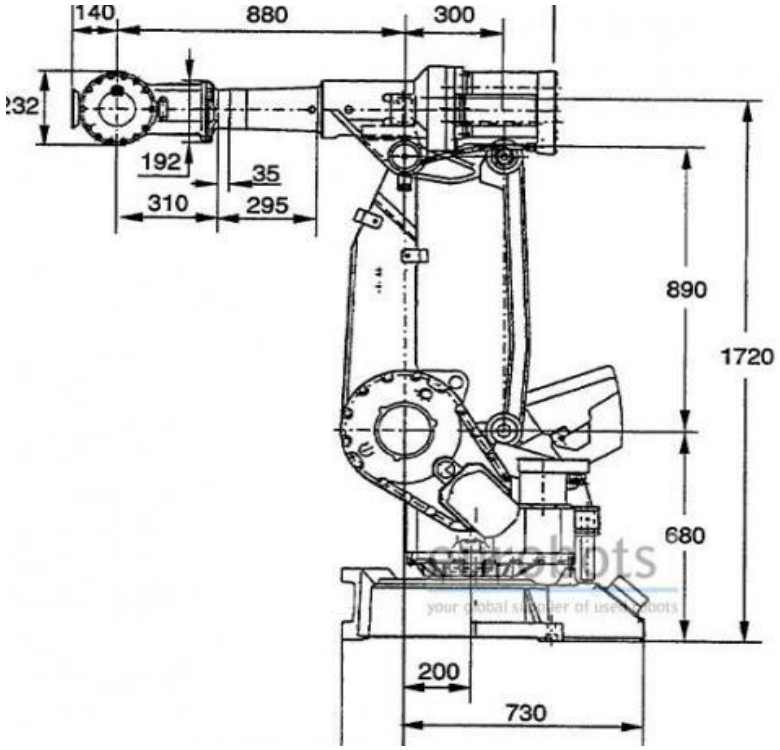


ABB LAYOUT AND MEASURMENTS

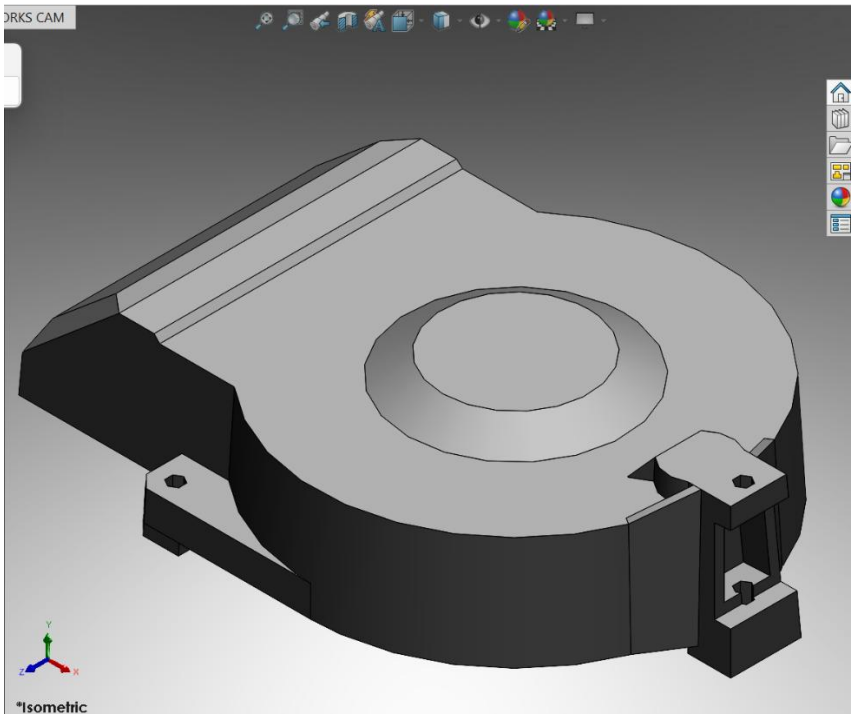


ABB JOINT ONE BASE PART 1

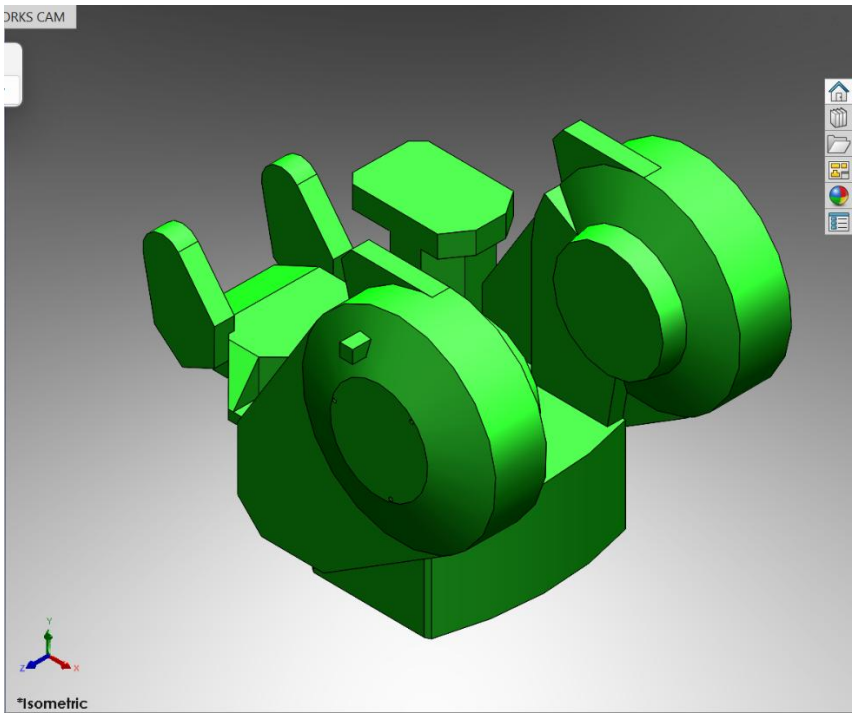


ABB JOINT 2 BODY PART 2

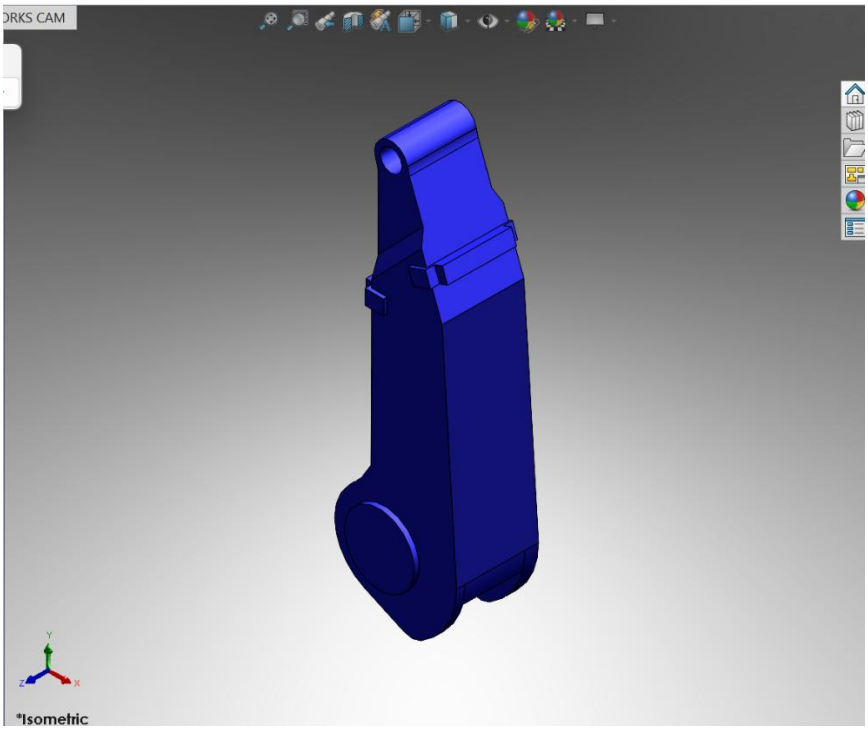


ABB JOINT 3 BODY PART 3

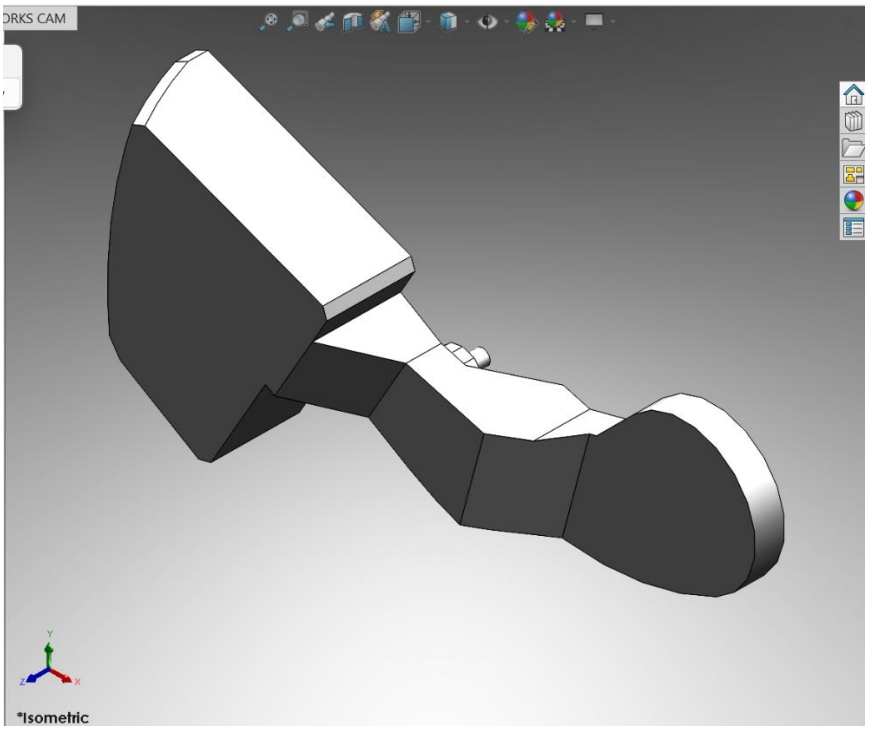


ABB COUNTERBALANCE PART 4

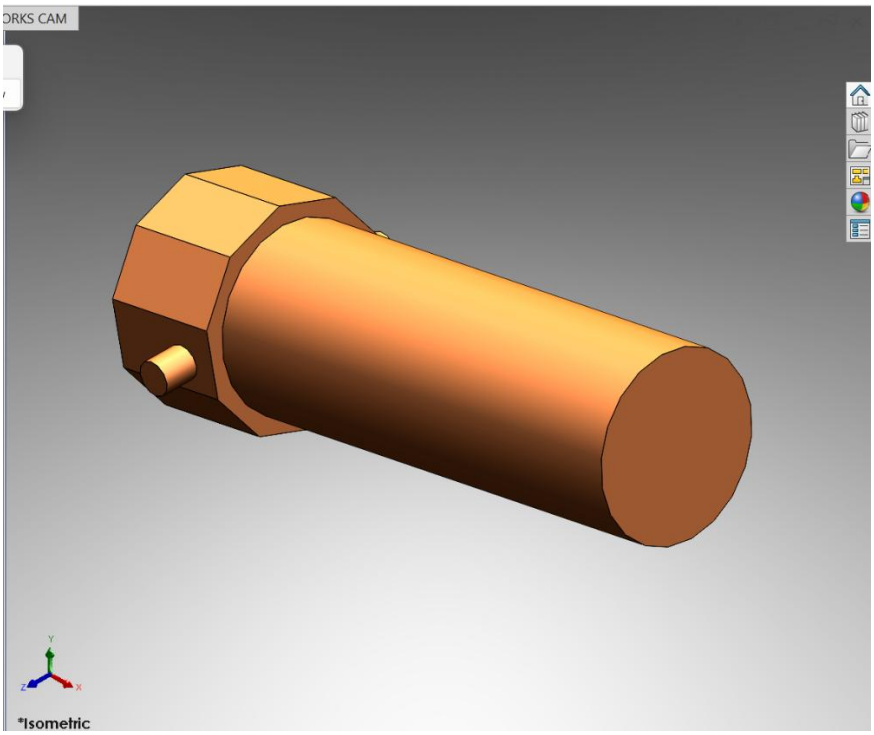


ABB ACTUATOR PART 5

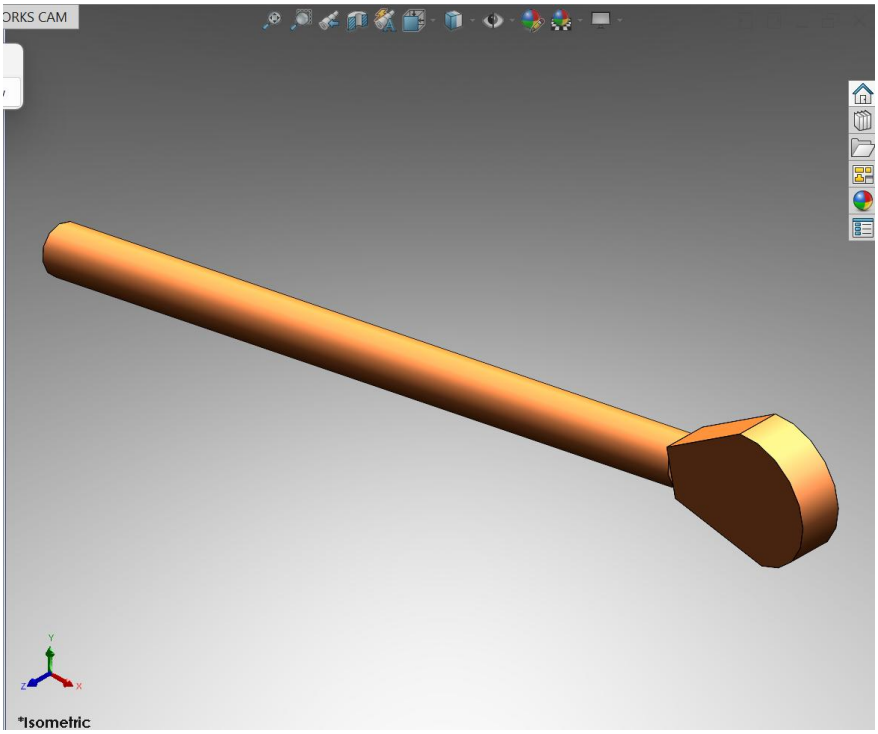


ABB ACTUATOR ROD PART 6

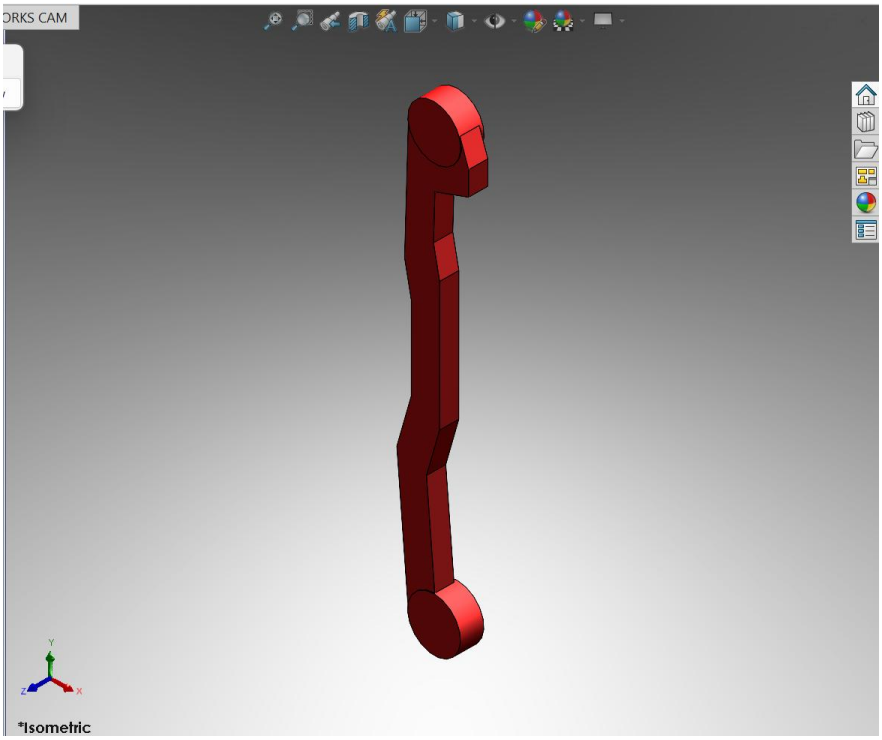


ABB ACTUATOR CONNECTING ROD PART 7

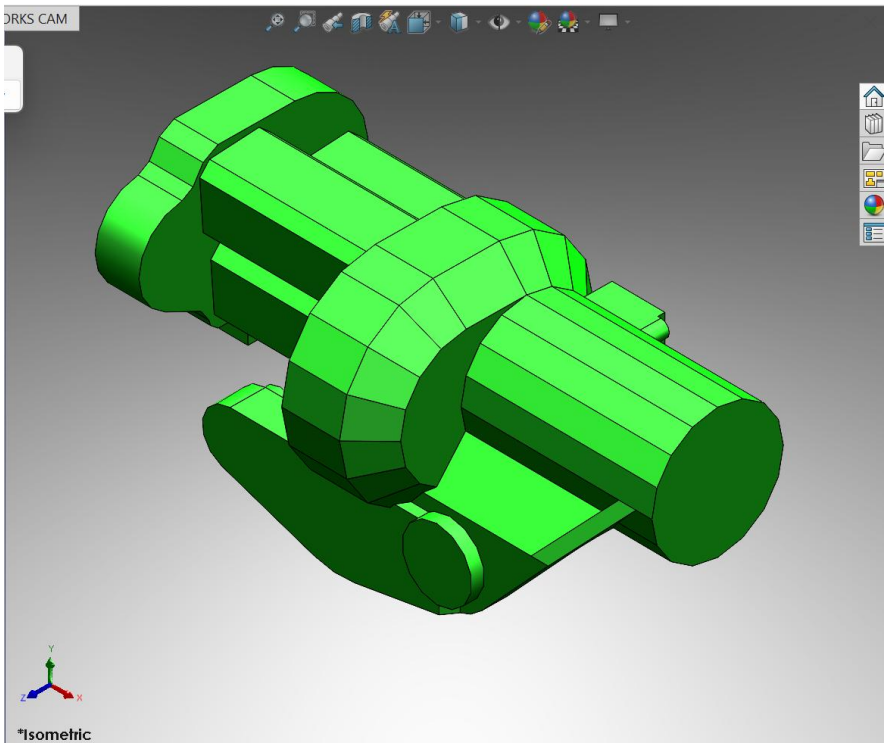


ABB JOINT 5 PART 8

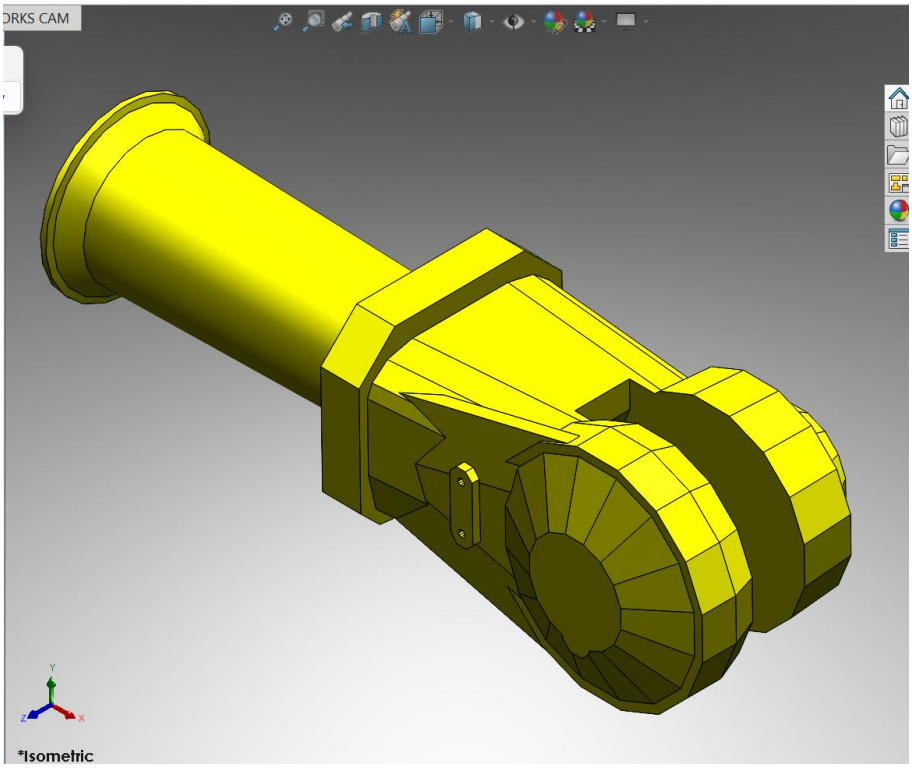


ABB JOINT 6 PART 9

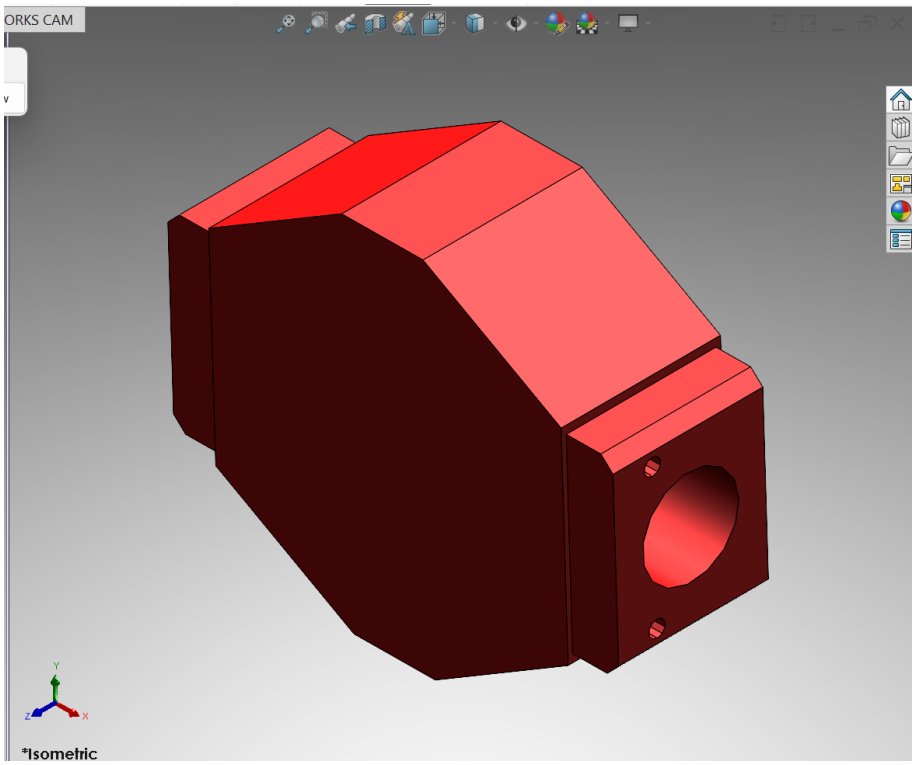


ABB END EFFECTOR HOUSING PART 10

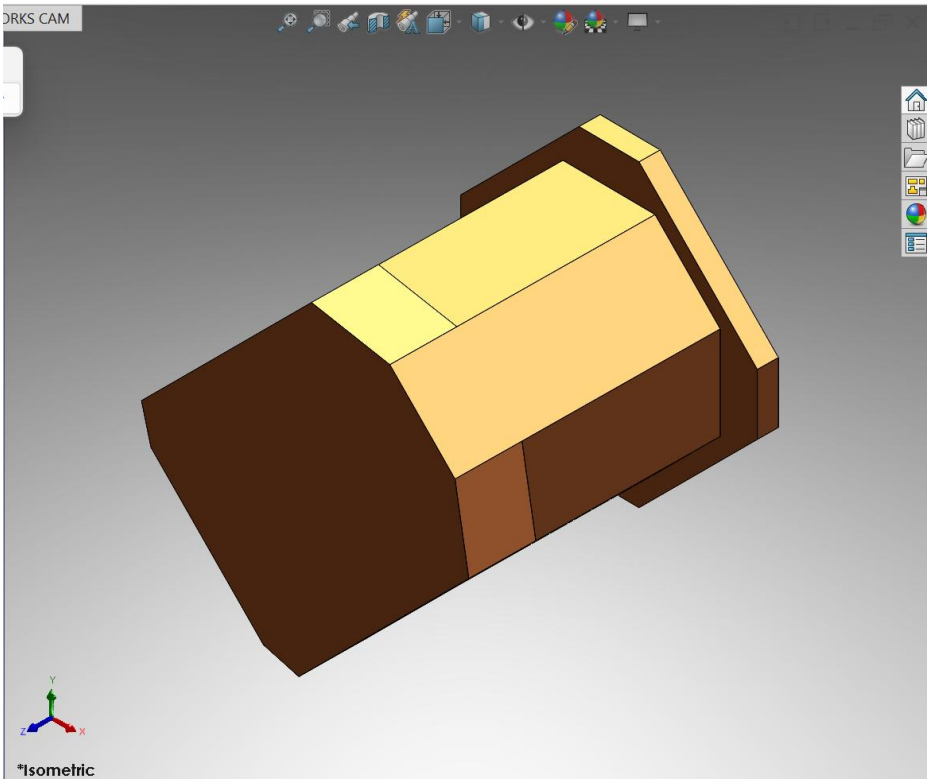


ABB LEFT HUB PART 11

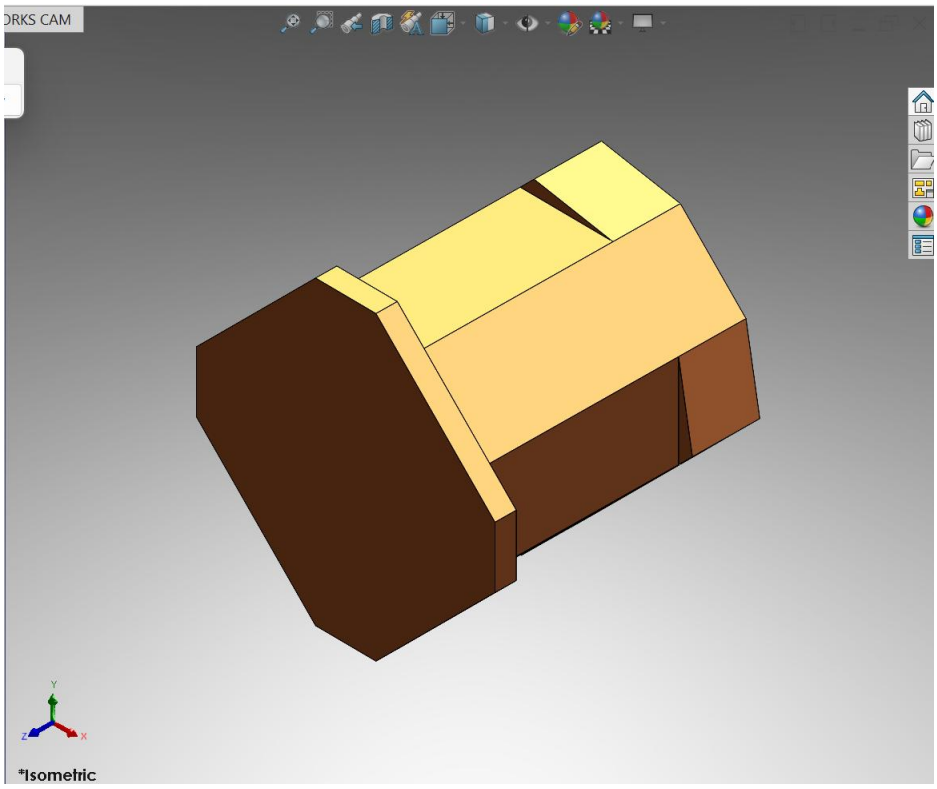


ABB RIGHT HUB PART 12

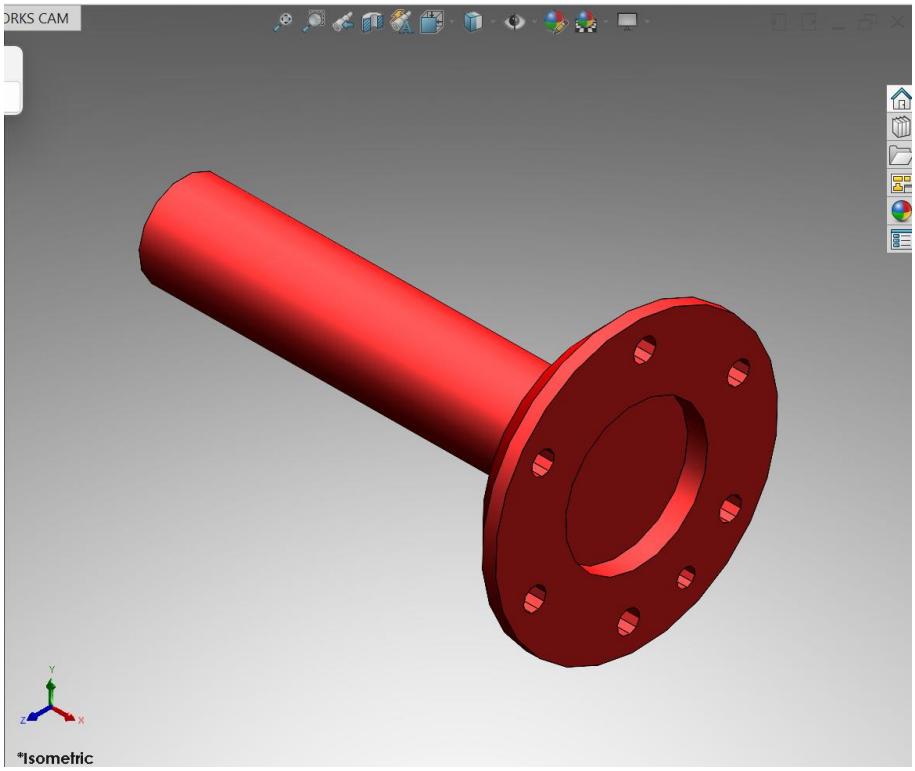


ABB END EFFECTOR PART 13

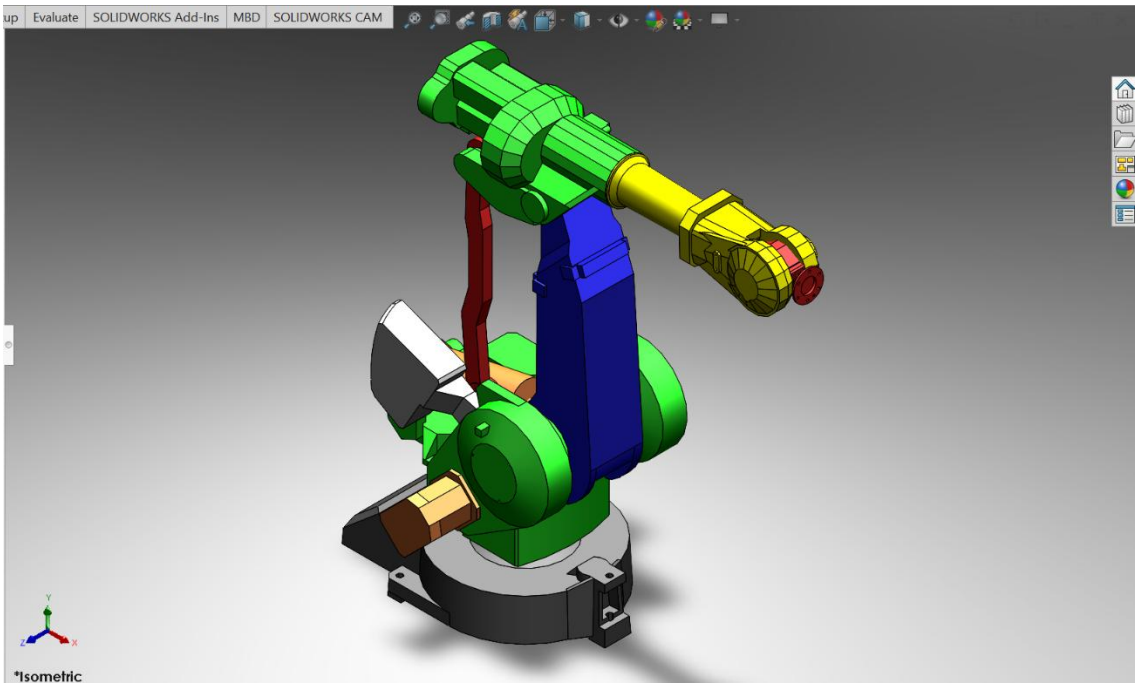
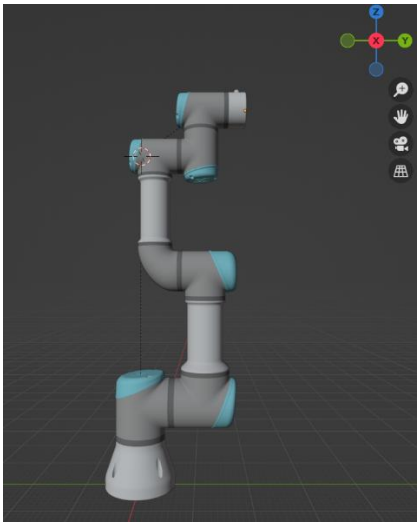


ABB ASSEMBLY

4.2. UR3 modelling

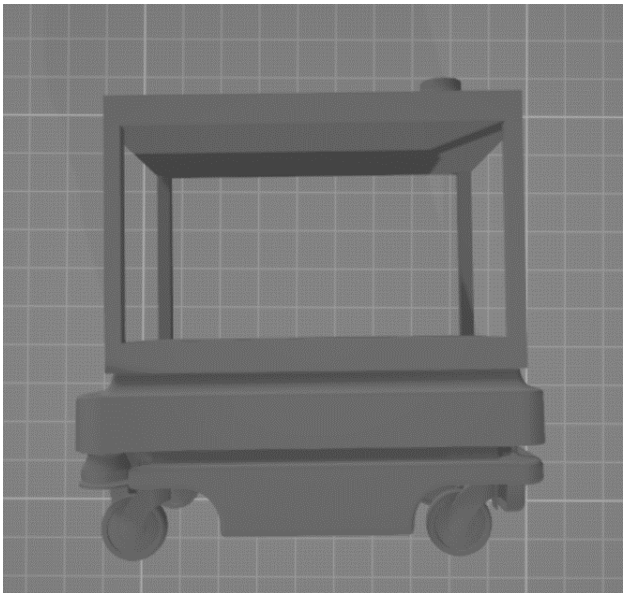
A thorough digital representation of the robot is created as part of the UR3 modelling process so that it may be integrated into different virtual and simulation settings. The UR3 is carefully modelled using CAD software like SolidWorks in order to represent its six-axis configuration and mechanical parts. In order to create FBX files compatible with Unity, this model is first converted to STL files and imported into Blender. This allows for precise and engaging simulations for tasks like path planning, collision detection, and operational testing in a digital twin environment. In the Project we are using 2 UR3 robots, one mounted to a Frame and the other mounted on MirMODlight Mobile robot.



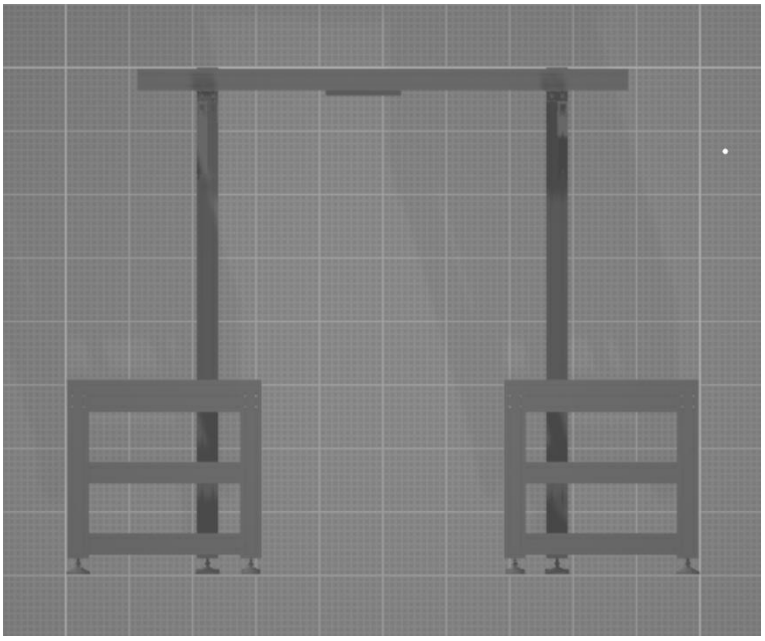
UR3 Robot

4.3. Mobile Robot MIR

The technique of modelling a mobile robot involves digitally simulating the robot with exacting details, emphasizing its lab environment navigation and interaction capabilities. The structural and mechanical features of the mobile robot are precisely modelled using CAD software such as SolidWorks. After that, these models are exported as STL files and imported into Blender to create Unity-compatible FBX files. Realistic lab-based mobility, obstacle avoidance, and task execution simulations are made possible by this smooth integration into the digital twin system.



MirMODLight (Mobile Robot)



UR3 Frame

4.4. LAB Layout

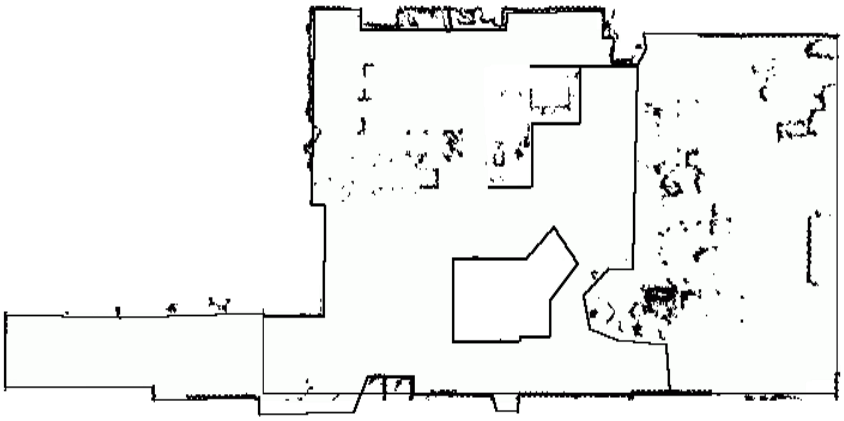
The process of building a complete digital model of the laboratory setting entails figuring out how the apparatus, workstations, and paths for the mobile robot are arranged in space. A thorough 3D model of the lab is made with SolidWorks, guaranteeing precise measurements and locations. After being exported as STL files, Blender is used to process the models and produce Unity-compatible FBX files. This digital structure is essential for streamlining processes, guaranteeing effective navigation and operation inside the lab space, and replicating the interactions between the robots and their surroundings.

4.4.1. Functional area of the lab

The lab's functional space is cleverly split into two halves to enable effective robot presentation and administration. The first section is devoted to displaying the mobility of the mobile robot as it moves about the lab and has an open layout with no barriers. This area is free to roam around, which makes it perfect for monitoring and enhancing the robot's navigation and path planning. All robots, including the ABB IRB 4400 and UR3, will be visualized in the second room, which will have intricate joint animation displays. This section is essential for keeping an eye on the robots' exact motions and interactions while offering a thorough overview of their coordination and functionality in the lab.

The primary display zone for the mobile robot is the unobstructed, plain surface area of the lab. It is possible to navigate and demonstrate movement in this open space with ease. One lab corner is designated as the starting point for the mobile robot's global axis system, which is specified in relation to the lab's coordinates. The mobile robot has a sensor installed at this corner that allows it to track its orientation and position with accuracy. This sensor records the robot's yaw angle and position coordinates continuously, allowing for accurate localization inside the lab. After that, the collected data is transmitted over MQTT and Node-RED, allowing for real-time animation and viewing of the robot's motions within Unity. This

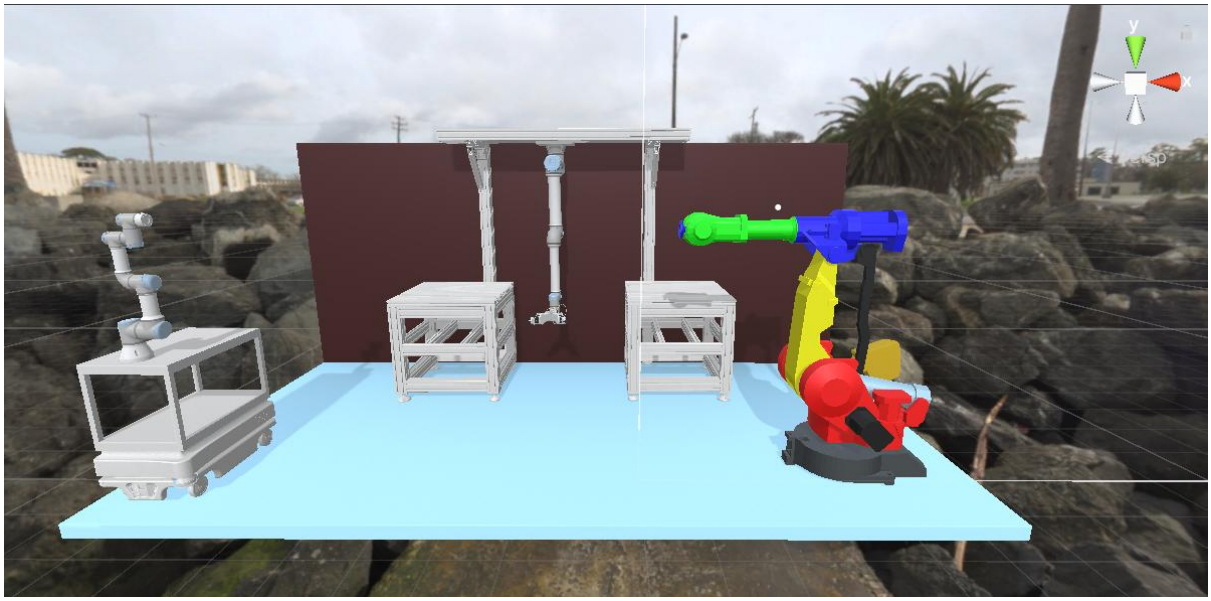
configuration not only displays the mobility of the robot but also demonstrates how real-time data processing and sophisticated communication protocols are integrated into the lab's digital twin environment.



Lab layout according to MirMODlight (Mobile Robot)

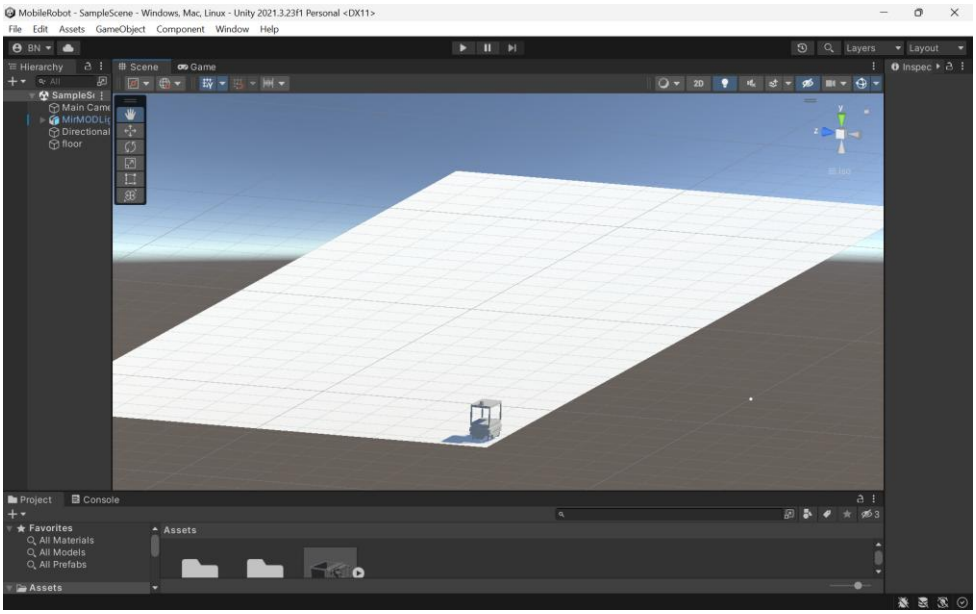
4.4.2. Simulation Environment in Unity

To guarantee convenience and thorough visualization, the Unity simulation environment was built in two stages. Two distinct environments were made in the initial stage. The initial setting is made to accommodate every robot in a single frame, making it possible to see how they cooperate and move as a team. This environment shows the coordination and interaction between many robots and offers a comprehensive picture of the overall robotic system.



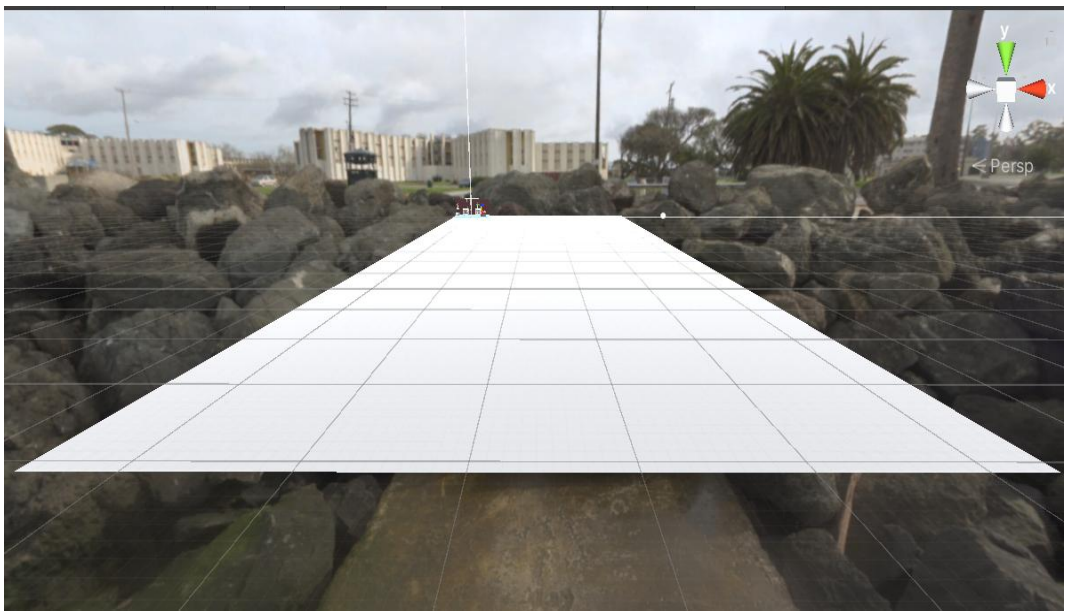
Multi robot unity environment

Specifically, the second environment is the lab area where MirMOBlight, the lone mobile robot, is present. The goal of this configuration is to precisely replicate the yaw angle and position of the mobile robot as it moves throughout the lab. To guarantee realistic motion and accurate data representation, the mobile robot may be isolated in its own environment and subjected to precise adjustments and calibrations.

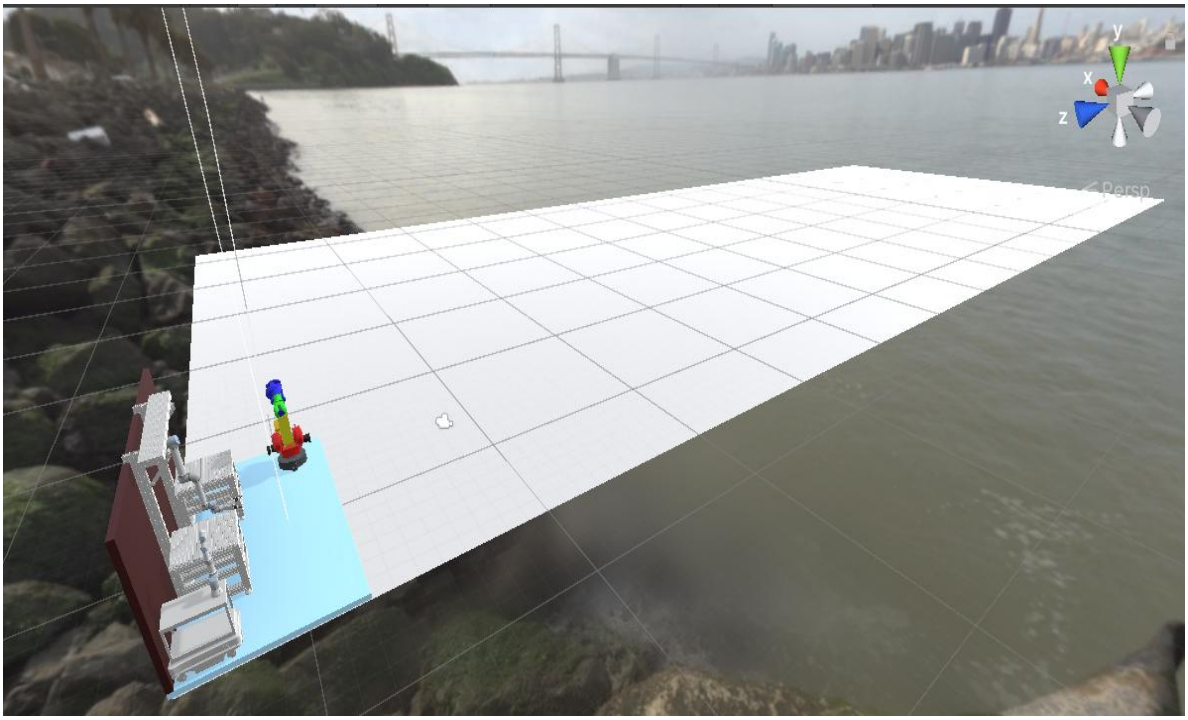


Unity environment for mobile robot

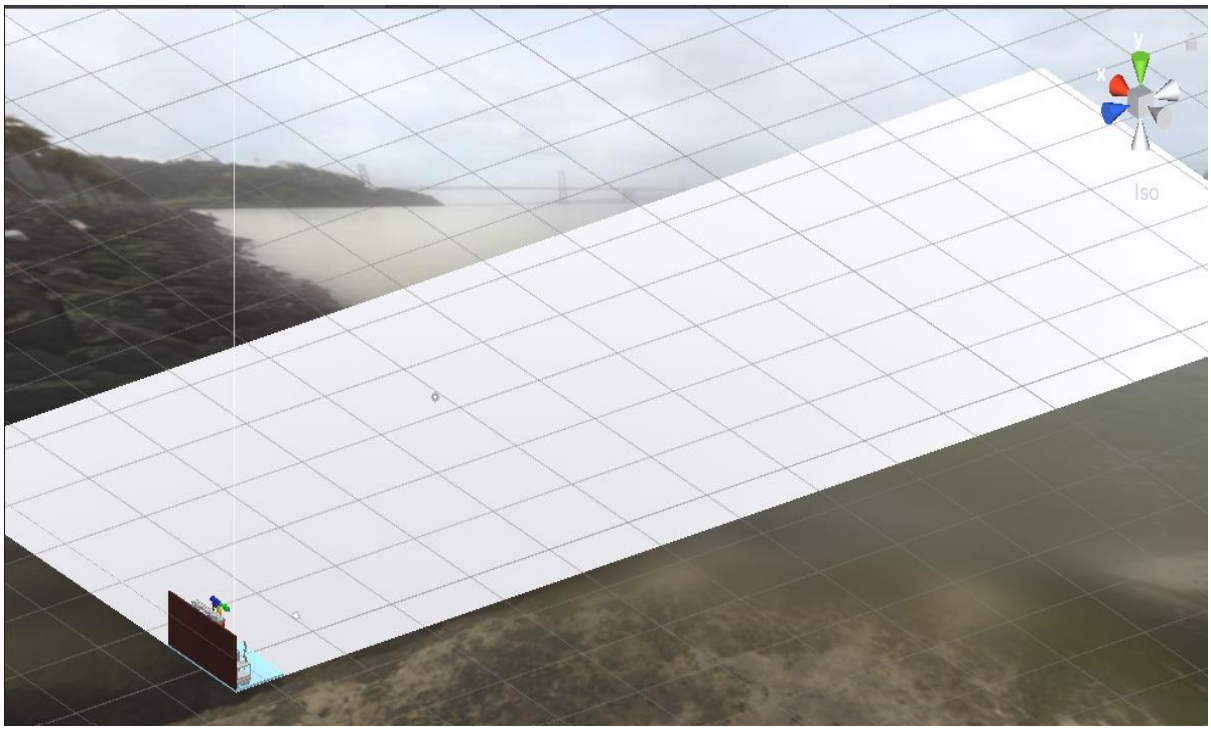
These two Unity environments were smoothly combined in the second stage. This integration makes it possible to create a thorough simulation that replicates the precise movements of the mobile robot inside the lab as well as the intricate visualization of every robot cooperating with one another. This two-phase method guarantees the simulation environment's flexibility and detail, offering a strong foundation for evaluating and displaying the lab's digital twin.



Unity Environment with Lab Floor and All Robots front view



Unity Environment with Lab Floor and All Robots Isometric view 1



Unity Environment with Lab Floor and All Robots Isometric view 2

Unity Environment with Lab Floor and All Robots top view

4.4.3. Unity Kinematics Defining

Unity Kinematics works on the global and local Reference frames, global and Euler Reference frames. Based on this concept every object or model in unity is defined by 2 reference frames (coordinate system), one object reference frame and two global reference frame.

Global Reference Frame

Within Unity, the scene's total coordinate system is referred to as the global reference frame. All objects in the scene use this fixed coordinate system as their common reference. The position, rotation, and scale of objects with respect to the scene's origin point—typically located at coordinates (0, 0, 0) — are defined by the global reference frame.

- **Position:** An object's location within the overall scene is its global position. Three coordinates (X, Y, Z) are used to represent this, and they show how far it is from the origin on each axis.
- **Rotation:** The object's orientation with respect to the world axes is specified by the global rotation. Typically, quaternions or Euler angles are used to describe this.
- **Scale:** An object's size in respect to the global coordinate system is determined by its global scale.

By allowing items to be placed and moved consistently throughout the scene, the global reference frame helps to guarantee that every element is in the proper location in relation to every other element.

Local Reference Frame

Conversely, the local reference frame is unique to every single object. If the object has no parent, it is defined in relation to itself. If not, it is defined in relation to the object's parent.

More precise control over an object's transformations (position, rotation, and scale) is possible with the local reference frame.

- **Local Position:** The item's location in relation to its parent object is indicated here. The child's global position changes when the parent moves, but its local position stays the same.
- **Local Rotation:** The orientation of an object with respect to its parent is defined by its local rotation. While the local rotation stays constant, changes to the parent's rotation will have an impact on the child's global rotation.
- **Local Scale:** The object's size in relation to its parent's scale is indicated by the local scale. The global scale of the child will change in response to changes made to the parent's scale.

Developers can establish hierarchical relationships between items and manage complex structures and animations more easily by using the local reference frame. In a robotic arm, for instance, every joint may have a local reference frame that is relative to its parent joint. This feature enables accurate control of specific segments while preserving the overall structure.

Comprehending the distinction between global and local reference frames in Unity is crucial for precise object manipulation within a three-dimensional environment. When animating robots, for example, the local reference frame permits fine control over specific joints and parts, while the global reference frame guarantees proper movement of the robot within the laboratory. This dual-frame method makes realistic and accurate animations possible, which is essential for producing successful digital twins and solution.

4.4.4. ABB Unity Kinematics

The Euler coordinate system idea is used in Unity to construct the ABB robot. The robot's movements may be precisely controlled and animated with this method. The ABB robot's components are defined hierarchically in this implementation, with each joint being represented as an empty game object that serves as an Euler axis for the parts that it is connected to.

The hierarchy in this arrangement is built up to match the robot's actual physical design. The pivot point for rotation and movement of each joint in the ABB robot is a specific empty game object in Unity. These empty game objects, also known as Euler axes, are positioned in a way that corresponds to the robot's actual joints.

Hierarchical Structure

- Base: The robot's base acts as the hierarchy's root. It serves as the main point of reference from which all other components are connected.
- Joint 1: A child of the base, an empty game object represents the first joint. This item serves as the robot's initial segment's pivot.
- Segment 1: Joint 1 is where the first segment's physical component is attached. The changes in this segment are in relation to Joint 1.
- Joint 2: A child of Segment 1 is an empty game object that represents the second joint. For every succeeding joint and section of the robot, this configuration is repeated.

Euler Axis

Euler rotation axes are set up in each joint's empty game object to specify the movement and rotation of the associated segment. For a correct simulation of the robot's articulation, these axes are essential.
As an illustration:

- Joint 1: To replicate base rotation, it rotates around the Y-axis.
- Joint 2: Simulates movement of the first segment by rotating around the X-axis.
- Joint 3: In a similar manner, it revolves around its own axis to regulate the motion of the subsequent segment.

Unity Implementation

The Euler axes in Unity are specified in each empty game object's Transform component. To produce the necessary motion, the rotation values are supplied in degrees and changed accordingly. The robot's joints can simulate movements found in the actual environment by animating these rotation.

The designated axes for every component of the ABB robot are shown in the picture below. The Euler coordinate system in conjunction with this hierarchical structure guarantees that the movements of the robot are faithfully and realistically simulated within the Unity environment.

By controlling each joint independently, developers can create the intricate animations and accurate simulations required for jobs like robotic assembly, maintenance, and operational training.

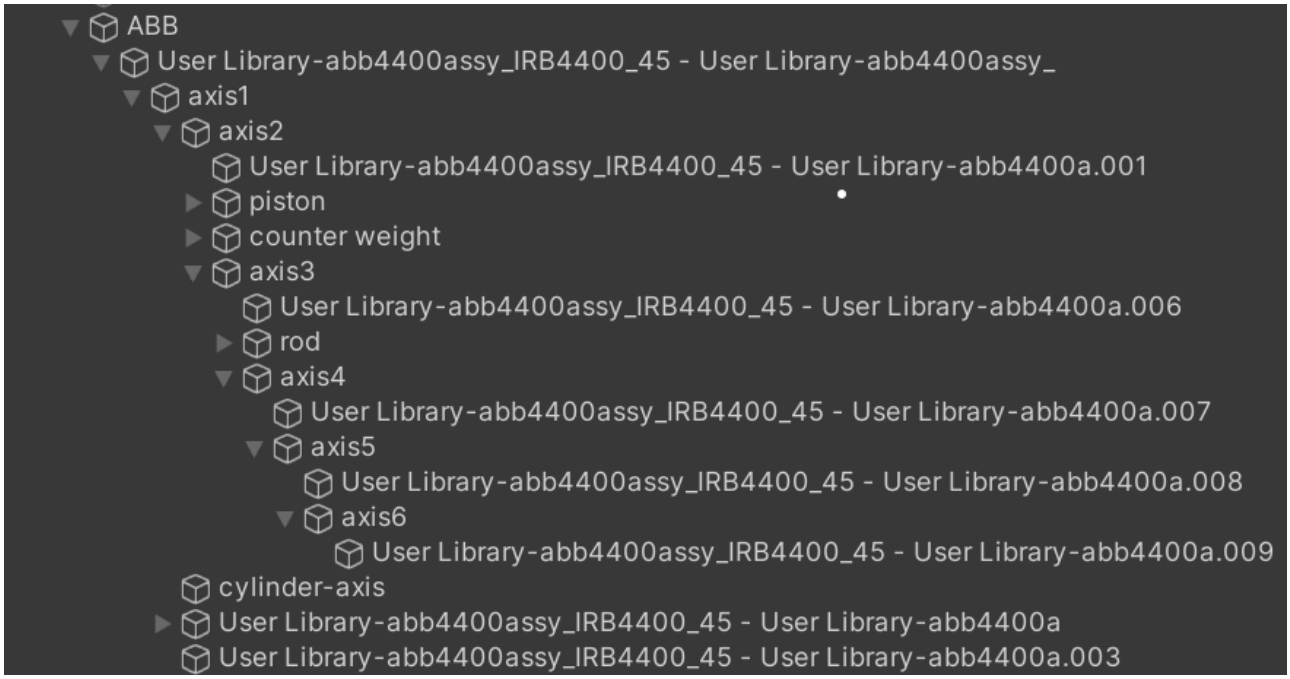


ABB Unity structure

4.4.5. Mobile Robot Unity Kinematics

To accomplish precise control and mobility within the Unity environment, we combine global, local, and Euler reference frames for the mobile robot (MirMODLight). With this method, the UR3 robot, fixed on the mobile platform, travels in relation to the mobile robot while the mobile robot navigates the floor coordinate system.

Global Reference Frame

The general coordinate system of the environment is represented by Unity's global reference frame. It is employed to regulate the mobile robot's orientation and location across the whole lab configuration. As the main reference point for all objects in the scene, the origin of the global reference frame is usually positioned at a specific corner of the lab.

- Mobile Robot (MirMODLight): The global coordinate system is used to control the location and orientation of the mobile robot. This guarantees that its movements are faithfully captured in the spatial context of the lab.

Local Reference Frame

Within the global coordinate system, each individual item has its own local reference frame. The local reference frame is essential for preserving the location and orientation of the UR3 robot mounted on the mobile robot with respect to the mobile platform.

UR3 Robot: The mobile robot serves as the reference point for the UR3 robot's motions. The UR3 robot's local reference frame makes sure it moves in sync with the mobile platform as the mobile robot moves around the lab.

Euler Reference Frame

The UR3 robot and the mobile robot both have rotations that are controlled by the Euler reference frame. The UR3 robot's joints and segments are all determined by Euler angles, which enables accurate rotational control.

The movement of the mobile robot is managed by the Euler angles, which are used for lab navigation.

Joints of the UR3 Robot: The UR3 robot's joints are each represented in Unity as empty game objects that serve as an Euler axis for the corresponding components. Accurate simulation of the articulations of the UR3 robot is made possible by this hierarchical framework.

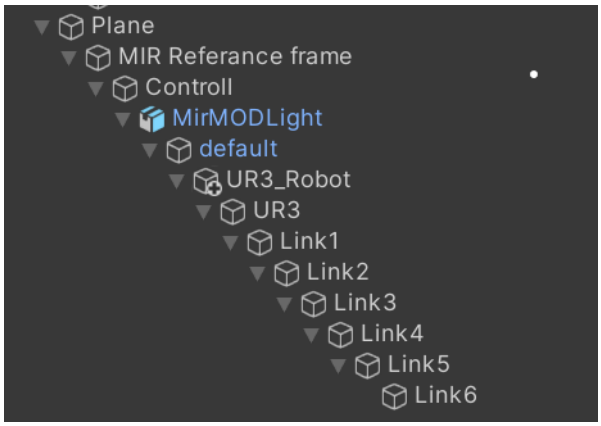
Implementation in Unity

The Transform component in Unity is used to determine the global position and orientation of the mobile robot. Nestled under the mobile robot's Transform, the UR3 robot's own Transform components control its local movements and rotations.

- Control of Mobile Robot: The position and orientation of the mobile robot inside the global reference frame of the lab are managed by its transform.

- UR3 Robot Control: The empty game object at each joint serves as an Euler axis to control how the connected segments rotate. Node-RED and MQTT data are used to update the joint angles (J1, J2, J3, etc.).

The design of the UR3 and mobile robots in relation to the floor (Plane) is depicted in the graphic below, which also shows how the interactions between the global, local, and Euler reference frames allow for precise and synchronized movements. This methodical approach guarantees that the UR3 robot and the mobile robot can function together in the Unity environment, offering a realistic simulation for a range of robotic applications and tasks.



Mobile Robot Unity structure

5. Digital Twin Communication architecture

To guarantee smooth communication between real robots and their Unity virtual version, the Digital Twin technology was created. Several communication protocols are integrated into the architecture, and Node-RED serves as a middleware to aggregate and process data from various robotic systems. A structured pipeline ensures effective transmission and real-time synchronization of the data flow.

The ABB, UR3, and MIR mobile robots all transmit data using different communication protocols, as seen in the communication architecture diagram:

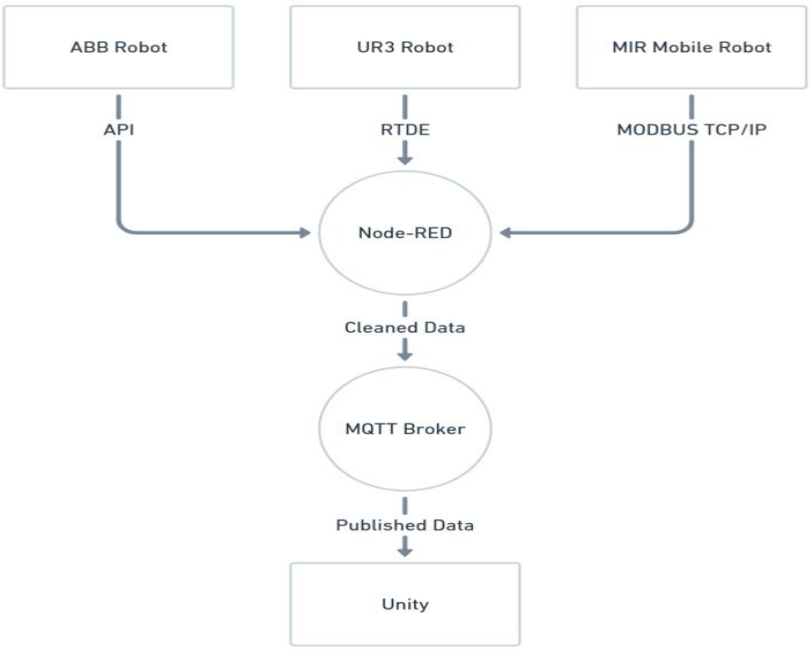
- **ABB Robot** → Communicates using **API** to provide joint and positional data.
- **UR3 Robot** → Uses **Real-Time Data Exchange (RTDE)** to stream real-time state information.
- **MIR Mobile Robot** → Transmits position and orientation data using **MODBUS TCP/IP**.

Node-RED receives all incoming data and acts as a universal plug-in to collect, preprocess, and standardize the data. This stage guarantees that the unprocessed data from various robots is sorted and organized into a common format.

The data is sent to the MQTT Broker for processing and publication as structured topics. After that, Unity subscribes to these MQTT topics and gets the most recent information

instantly. By using this data to animate the Digital Twin, the Unity environment makes sure that the robots' virtual representations faithfully replicate their real-world counterparts.

An integral part of the Digital Twin structure, this modular and scalable architecture facilitates dependable communication, real-time monitoring, and effective robot coordination.



Communication network architecture

6. Implementation of the Digital Twin

6.1. Data Collection and Processing

To provide an accurate and up-to-date simulation of the robots' operations, data gathering and processing are essential phases in the development of the digital twin. Node-RED and Unity are used in a multi-layered manner to implement this process.

In Node-RED, we establish three separate data flows, each in charge of gathering information from various robots. Depending on the particular API that each robot uses, the data that is received from them is either in the JSON or XML format. Some robots may use XML, however the ABB robot uses an API that generates JSON to convey its data. The data is transformed into raw labelled data after it is gathered. In order to guarantee that the data is consistently formatted and to make processing and analysis easier in the future, this transformation is essential.

The tagged data is uploaded to a MQTT broker after transformation. Multiple clients (in this case, the Unity application) can subscribe to and receive the data in real-time through the MQTT broker, which serves as a central hub for data distribution. By using this method, the

data flow is guaranteed to be scalable and effective, meeting the changing needs of the digital twin.

Unity Data Processing: UI scripts and game object control scripts are the two categories of scripts that Unity uses to handle the data gathered from the MQTT broker.

UI Scripts: Unity's UI scripts handle the gathering and preliminary processing of data. The MQTT topics where the tagged data is released are subscribed to by these scripts. The data is received by the UI scripts, which then process it and format it so that the game objects may use it with ease. For instance, the ABB robot's joint angle data is processed and allocated to variables denoted J1, J2, J3, and so on, which correspond to the robot's individual joints.

Game Object Control Scripts: These scripts take over from UI scripts in order to control how the robots move physically inside the Unity world. For example, every joint on the ABB robot is connected to a unique control script. These control scripts make sure that the virtual robot moves precisely in line with the data received from the real robot by updating the joint angles in real-time depending on the processed data from the UI scripts.

The implementation of an organized method for gathering and analysing data allows us to seamlessly integrate the digital twins of the real robots with Unity. In addition to providing real-time visualization and control, this configuration paves the way for future improvements and integrations within the digital twin environment.

6.1.1. Data Acquisition Techniques:

Data acquisition, data sharing, or the digital building blocks of edge computing and cloud storage technologies are essential elements for constructing a reliable Digital Twin. Here we describe how data from a variety of robotic systems are collected via specialized communication protocols and middleware.

1. Robotic Systems and Communication Protocols

For transmitting real-time data to the Digital Twin, the three main robotic systems utilized in this project—the ABB IRB 4400, UR3, and the mobile robot—each have their own connection protocol:

ABB IRB 4400 Robot:

- Data, including joint angles (J1 to J6) and other operational parameters, are retrieved via the ABB API.
- To query the API and publish the data into the MQTT broker, Node-RED acts as middleware.
- **Testing the API endpoint:** Tools such as Postman are utilized to validate the data retrieved from the ABB API.
- **Sample data structure:**

  {

```
    "joint1": 30.5,
    "joint2": 45.2,
    "joint3": -12.8,
    "joint4": 0.0,
    "joint5": 20.5,
    "joint6": -10.0
}
```


UR3 Robots:

- The Real-Time Data Exchange (RTDE) is used by the UR3 robots, enabling quick transfer of joint angles and operational data.
- After processing the incoming RTDE stream, Node-RED publishes the data to the MQTT broker in a defined format.

Mobile Robot:

- Uses the MODbus protocol to send positional data (posix, positionZ) and orientation (yaw).
- The data is collected, transformed into standard JSON format in Node-RED, and then forwarded to the MQTT broker.

2. Middleware Integration with Node-RED

Node-RED serves as a versatile and scalable middleware layer, connecting robotic systems with the MQTT broker:

Flow Configuration:

- Data from each robot is processed through dedicated Node-RED flows.
- For instance, API nodes collect data from ABB robots, RTDE nodes manage UR3 communications, and MODbus nodes extract data from mobile robots.
- This ensures that the data is formatted uniformly for integration into the Digital Twin.

Real-Time Data Publishing:

- Node-RED publishes all robot data to the MQTT broker under specific topics. Examples include:
    - ABB motor joint angles: abbj1, abbj2, ..., abbj6
    - Mobile robot position: posix, positionZ
    - Mobile robot orientation: yaw

3. MQTT as the Core Data Transmission Hub

An MQTT broker serves as a centralized platform for data exchange, enabling real-time data integration into Unity for visualization and control:

Topic Structure:

- Data from each robot is published to uniquely configured MQTT topics, allowing easy identification and subscription.
- **ABB example:**

  Topic: abbj1
  Message: {"value": 30.5}

- **Mobile robot example:**
  Topic: posix
  Message: {"value": 10.0}

4. Real-Time Synchronization with Unity

In Unity, the ConnectionMgr script handles the reception of data from the MQTT broker and updates robot states accordingly:

Topic Subscription:

- Subscribes to relevant topics (e.g., abbj1, posix, etc.).
- Receives and decodes data for immediate use.

```
private void Client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
{
    string topic = e.Topic;
    string message = System.Text.Encoding.UTF8.GetString(e.Message);

    switch (topic)
    {
        case "abbj1":
            J1 = float.Parse(message);
            break;
        case "posix":
            posx = float.Parse(message);
            break;
    }
```

}

6.1.2. Data Processing Algorithums:

Efficient data processing is essential to converting raw data from robotic systems into meaningful insights for the Digital Twin. This process involves organizing, transforming, and synchronizing data to ensure accurate visualization and control within the Unity environment.

1.Preprocessing in Node-RED

Node-RED provides the first tier of data processing: gathering from various robotic systems, normalization, and further forwarding to an MQTT broker.

Standardization of Data: Data formats vary between robots:

- ABB robot data is provided via API in JSON format.
- RTDE streams are carrying UR3 robot data.
- The data of the mobile robot uses MODbus communication.

Node-RED converts this heterogeneous input to a homogeneous JSON format:

```
{
  "joint1": 30.5,
  "joint2": 45.2,
  "positionX": 10.0,
  "yawAngle": 15.0
}
```

Real-time publishing:

- The pre-processed data are published on MQTT topics depending on their origin and type, e.g. abbj1 for ABB joint angles or posix for mobile robot positions.
- Data publication frequency is set in accordance with the system requirement trade-off between real-time performance and network bandwidth.

2.Data Reception in Unity

Unity serves as the processing hub where real-time data from the MQTT broker is utilized in synchronizing the Digital Twin with physical systems.

Subscribe to MQTT Topics:
The ConnectionMgr script subscribes to relevant topics to receive updates:

```
foreach (string topic in topics)
{
```

client.Subscribe(new string[] { topic }, new byte[] { MqttMsgBase.QOS_LEVEL_AT_LEAST_ONCE });

}

Analyse Received Data:

The received data is parsed and assigned to static variables, ensuring that the most recent values are accessible across all Unity scripts.

3. Real-time Synchronization of Robotic Joints

Data processing is applied to robotic models within Unity for real-time synchronization:

Control Scripts for Joints:

Each joint is controlled by a dedicated script; for example, ABB_link_1 controls Axis 1, ABB_link_2 controls Axis 2. Then, the data of each joint is processed and applied to rotate the corresponding GameObject:

```
transform.localEulerAngles = new Vector3(
    0f, // X rotation
    (float)((-1) * ConnectionMgr.J1), // Y rotation
    0f  // Z rotation
);
```

Combining Data for Dependent Joints:

For dependent axes (e.g., Axis 3's rotation depends on Axis 2's state), the data is combined algorithmically:

```
float rotationZ = (float)(ConnectionMgr.J3 - ConnectionMgr.J2);
transform.localEulerAngles = new Vector3(0f, 0f, rotationZ);
```

4. Positional and Orientation Processing for Mobile Robots

The mobile robot's position and orientation data are processed to enable realistic movement within Unity:

- **Position Updates**:Positional data (posix and positionZ) is applied to the robot's GameObject for real-time navigation:

  ```
  transform.position = Vector3.MoveTowards(transform.position, targetObject.transform.position, moveSpeed * Time.deltaTime);
  ```

- **Yaw Angle Updates**:The yaw angle (yaaw) is processed to align the robot's rotation with its physical counterpart:

  *Vector3 newDirection = Vector3.RotateTowards(transform.forward, targetDirection, rotationSpeed * Time.deltaTime, 0f);*
  transform.rotation = Quaternion.LookRotation(newDirection);

5. Ensuring Smooth Animations

To avoid abrupt changes and ensure smooth animations, data is interpolated between successive updates:

- **Position Interpolation**: Position updates use linear interpolation to create smooth transitions, reducing jitter caused by rapid data changes:

  transform.position = Vector3.Lerp(transform.position, targetObject.transform.position, 0.1f);

- **Rotation Smoothing**: Quaternion slerp (spherical linear interpolation) is applied to create natural-looking rotations.

6. Error Handling and Data Validation

Robust error handling is implemented to ensure reliability:

- **Invalid Data Detection**: Data parsing includes checks for out-of-range or malformed values. Invalid data is ignored, and an error is logged for debugging.

  ```
  try
  {
     J1 = float.Parse(message);
  }
  catch (FormatException)
  {
     Debug.LogError($"Invalid data received for J1: {message}");
  }
  ```

7. Unified Data Flow

The combination of Node-RED, MQTT, and Unity scripts establishes a seamless data flow:

- Node-RED preprocesses and publishes data.

- MQTT ensures real-time delivery.
- Unity processes the data and applies it to animate robotic models and synchronize the Digital Twin.

This structured data processing pipeline ensures that the Digital Twin best represents the state of the physical robots. It enables real-time visualization, control, and analysis.

6.2. Communication Protocal Implementation

Implementing reliable communication protocols is at the core of building a functional and effective Digital Twin. Communication serves as the bridge that connects the physical robots to their virtual counterparts, enabling the system to operate in real time. This process involves managing the seamless exchange of data between multiple systems with diverse communication standards and ensuring that the Unity-based virtual environment accurately reflects the state of the physical robots. By ensuring this synchronization, the Digital Twin becomes a powerful tool for monitoring, simulation, and control. To achieve this, the system integrates cutting-edge technologies such as Node-RED, MQTT, and protocol-specific configurations tailored to the requirements of each robot. These technologies work together to establish a robust and flexible architecture, allowing for consistent data flow, rapid updates, and the ability to scale for future expansions or additional robots. This integration ensures that the Digital Twin not only mirrors real-world operations but also enhances functionality by enabling predictive maintenance, real-time visualization, and advanced decision-making capabilities.

6.2.1. Setting up Node Red

Node-RED, in its role as a middleware, serves as an interlink between the Digital Twins when it comes to the communication architecture. Because of its flexibility and visual programming environment it is seen as a very useful tool for the gap between the robotic systems and the MQTT broker. What the types of devices and machines are connected, Node-RED allows for a simple way to transfer data using the different protocols. It does that by providing the ability to preprocess data, standardize them and then routing data. Not only that but it is instrumental in unifying the diverse types of robotic systems in one framework.

The robots in the system are going from one to the other by their own communication standard, and Node-RED makes sure this data is transformed into a standardized JSON that is applicable for further processing. For example, an ABB IRB 4400 robot provides joint angle and operational data through an API. Node-RED makes use of the HTTP request nodes which are connected to the appropriate API in order to ensure that the necessary data is collected without loss and in a strictly regular manner. Likewise, the UR3 machines are floating the instantaneous number through the Real-Time Data Exchange protocol. Node-RED actualized the capture of these RTDE streams and provided their information in the form of JSON messages and are then ready for publication to the MQTT broker. As the mobile robot communicates using the MODbus protocol, it sends its position and orientation information. MODbus nodes present in Node-RED are used to capture them and preprocess to ensure their correctness bill for their final destination entry in the right MQTT topics.

The functions of debugging and monitoring are indistinctly associated with the Node-RED Cortex. When the testing stage was in process, the debug nodes of the robot were the main parts, which were used to check whether the data collected from the robots was theoretically accurate and reliable. Visualization of the data flow and real-time error detection by developers was facilitated by these nodes. Afterward, Node-RED added automatic retry mechanisms for failed API calls or MODbus queries, thus keeping the operation of the system unaffected in the case of short communication failures.

Robot administration and monitoring can be done by the platform, whereas Node-RED is employed as a flexible and scalable middleware. It becomes a channel between the robotic systems and the MQTT broker. Moreover, it provides the necessary data consistency, reliability, and integration into the Unity-based Digital Twin, which finally is connected to the MQTT broker.

6.2.2. Setting up MQTT Broker

The MQTT broker is central to the communication network architecture. It becomes the centre of all the messages exchanged between Node-RED and Unity. Because it uses a publish-subscribe paradigm, MQTT is able to ensure that data is sent out in a timely manner.

The broker builds up the data approximately according to the robots and the parameters transmitted. For instance, angles of joints from the ABB robot are grouped into such subtopics as abbj1, abbj2, etc., while the position of the mobile robot is posted under the topics ‘posix’ and ‘positionZ’. Such a structure makes it very easy to find and subscribe to the required information streams.

Therefore, an important feature of the broker also involves the possibility of choosing the level of QoS. In this implementation, QoS Level 1 ensures that each message has at least once delivery. This provides reliable data flowing through. The broker also handles retained messages, which can be used to avoid possible gaps in synchronization, meaning that a newly subscribed subscriber gets the most recent values immediately.

Broker Configuration:

- The Mosquitto MQTT broker is deployed, supporting Quality of Service (QoS) levels to ensure reliable data delivery.
- Retained messages are used to provide new subscribers with the latest state of robots immediately upon connection.

Topic Structure:

1. Data is organized into hierarchical topics to distinguish between different robots and their parameters:
    - ABB Robot:
        - abbj1, abbj2, ..., abbj6 (joint angles)
    - Mobile Robot:

        - posix (X-axis position)
        - positionZ (Z-axis position)
        - yaaw (yaw angle)
    - UR3 Robot:
        - ur3j1, ur3j2, ..., ur3j6 (joint angles)

6.2.3. Configuring Robot Communication

For configuring API of ABB, we initially used Robot studio software for generating artificial data and used the defined API address to get information of the joint angles of all 6 axis of ABB. To check the API address, we used postman software. Here we can visualize the data format we receive from ABB. In postman we can manually send a single request to the API address and receive data at the instant. Requesting again will refresh the data. This process is automated using node red sending a request and receiving data at your desired speed.

Every robot needs its specific configuration for the Digital Twin to build on its communication protocol. For this, ABB IRB 4400 uses API; it continuously shows the angles of the joints and all other states. During the setup, some tools, like Postman, were used in testing the API endpoints to see if all the data obtained was valid and complete. Once tested and validated, this same information was directly fed into Node-RED for publishing into the MQTT broker.

ABB IRB 4400:

- An API endpoint is configured to retrieve joint angles and operational parameters.
- Postman is used to validate API endpoints, ensuring accurate data retrieval.
- Data is requested and received in JSON format:

```
{
   "joint1": 30.5,
   "joint2": 45.2,
   "joint3": -12.8,
   "joint4": 0.0,
   "joint5": 20.5,
   "joint6": -10.0
}
```

The UR3 robots are dependent on RTDE for data exchange. These robots stream data continuously and will keep updating their joint positions in real time. The incoming RTDE

packets are processed by Node-RED, which translates them into a JSON format that Unity is able to parse easily.

In the case of the mobile robot, the MODbus protocol was used. In this way, positional data and orientation data were gathered. The Node-RED MODbus nodes were tasked with extracting and preprocessing that data in preparation for MQTT transfer.

6.2.4. Ensuring Reliable Data Transmission

It was important during development that this communication system would be robust. Several mechanisms have been put in place that allow for reliable data flow and real-time synchronization. The use of retained messages in MQTT was one of the main measures. It would ensure that Unity always had the latest state of the robots, even in the case of small communication breaks. Moreover, Node-RED was configured to retry a failed API call or MODbus query, which minimizes the chance of critical data not being received. Another important aspect was error handling. At the Node-RED level, filtering out invalid data, such as out-of-range values or incomplete messages, prevented disruptions in Unity. Within Unity itself, exception handling was handled within scripts to log and manage errors without halting operations.

6.3. Unity Integration with C# Scripting

Unity integrated with C# scripting forms a critical component of the Digital Twin system for real-time visualization, control, and synchronization of physical robots within a virtual environment. Unity is used as the platform to render and animate the robotic systems, while C# scripts are used to communicate with the MQTT broker to receive data that has been processed and published by Node-RED flows. This combination ensures that the Unity-based Digital Twin mirrors the operations of physical robots with high accuracy and responsiveness.

Unity's role is not confined to rendering robot models but extends to providing a solid platform for integrating real-world data into simulation workflows. The Digital Twin system will utilize Unity's powerful 3D rendering and custom C# scripting to provide a seamless interface for data visualization, troubleshooting of robotic movements, and testing operations before applying them on physical systems.

6.3.1. Node Red Flows

Node-RED plays a central role in the pipeline of communication, acting as the middleware that bridges data from the various robotic systems to Unity. Its role in preprocessing and organizing data before publishing to MQTT is to ensure that Unity gets clean, structured, and standardized inputs. This section will detail how Node-RED flows are implemented in order to enable Unity integration, with particular focus on specific flows that are developed for different robotic systems.

Purpose of Node-RED in Unity Integration

Node-RED will be used within the Digital Twin architecture for decoupling the data acquisition and processing layer from Unity to let the latter focus only on rendering and animations. Communication is done by each robotic system-ABB robot, UR3 robots, or mobile robot-each using its own protocol. These diverse data streams are captured, processed into a consistent format, and published to respective MQTT topics. By doing this, Node-RED ensures that Unity will communicate with a standardized data structure, regardless of the lower-level protocol being used.

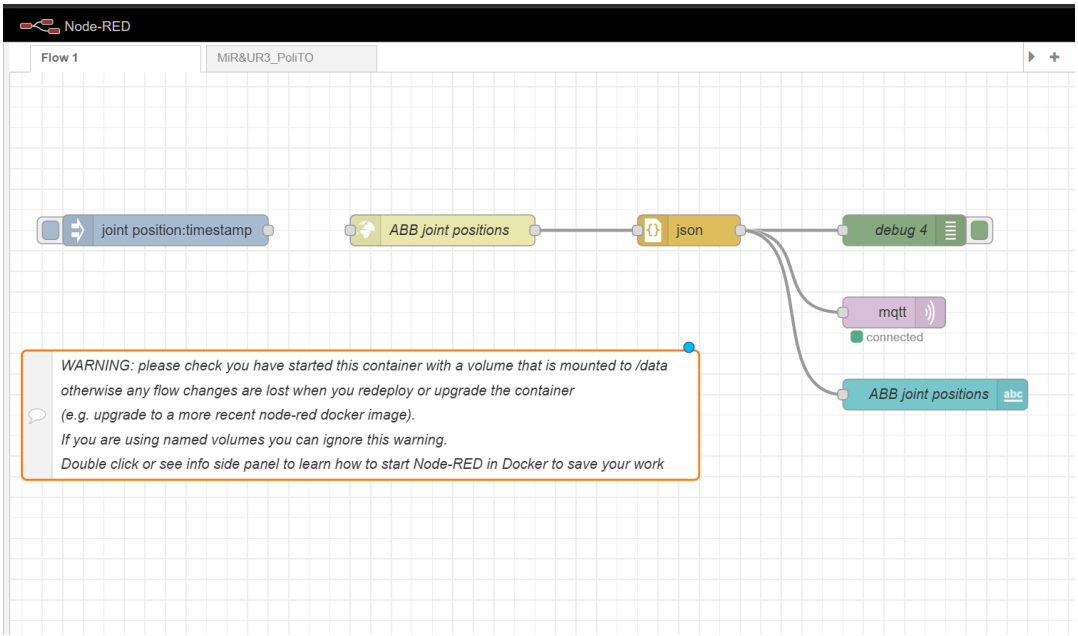
The flexibility of Node-RED also allows for live manipulation and debugging of the data streams before they get to Unity. Debug nodes are placed at strategic locations to monitor data flow; this ensures that errors and inconsistencies in the raw data are caught at an early stage in the data pipeline. This robust way of handling data minimizes issues downstream and ensures smooth visualization and synchronization in Unity.

Key Node-RED Flows

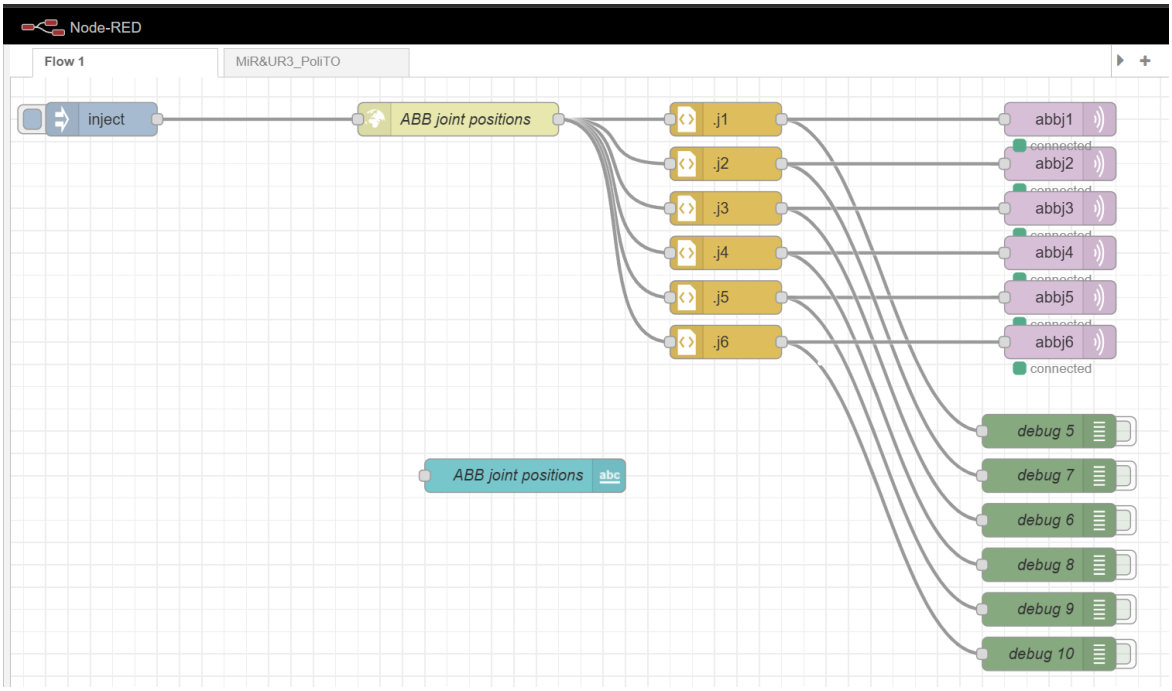
The integration of Unity with robotic systems relies on several Node-RED flows, each tailored to specific tasks. These flows process and publish data to MQTT topics, which are subsequently consumed by Unity through C# scripts.

1. **Flow for ABB Robot Joint Data**:

- This flow collects real-time joint angle data from the ABB IRB 4400 robot. An HTTP request node queries the ABB API at regular intervals, retrieving a JSON payload containing the robot's joint angles (J1 to J6).
- data is split into individual values, each corresponding to a specific joint, and published to separate MQTT topics (abbj1, abbj2, etc.).
- nodes are used to validate data accuracy at every stage of the flow, ensuring that Unity receives correct and consistent data.



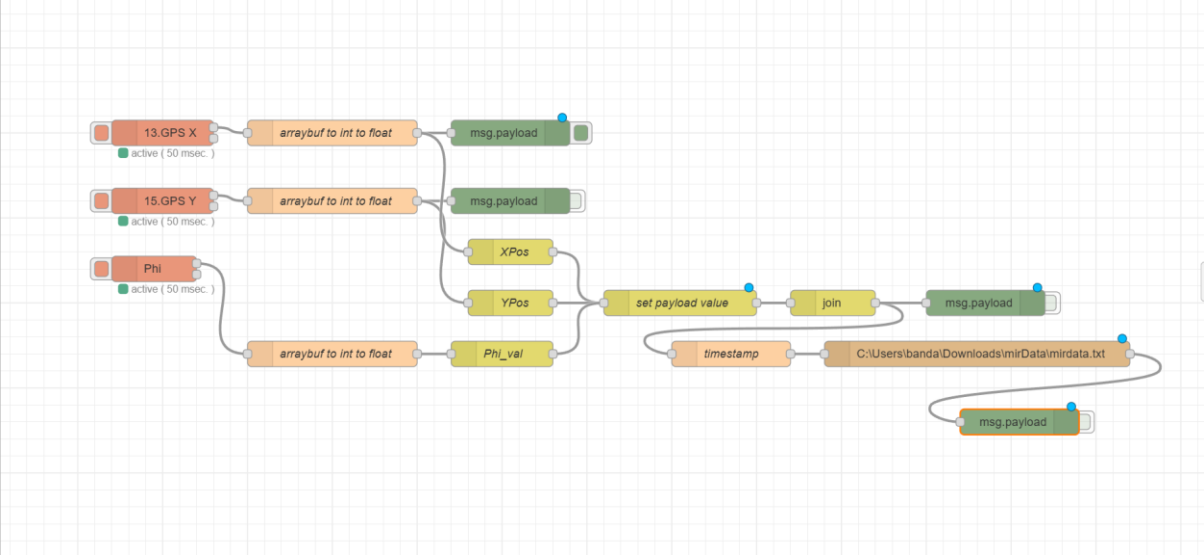
Node Red Structure for ABB data processing



Node Red flow for ABB Joints data

2. **Flow for Mobile Robot Position and Orientation**:

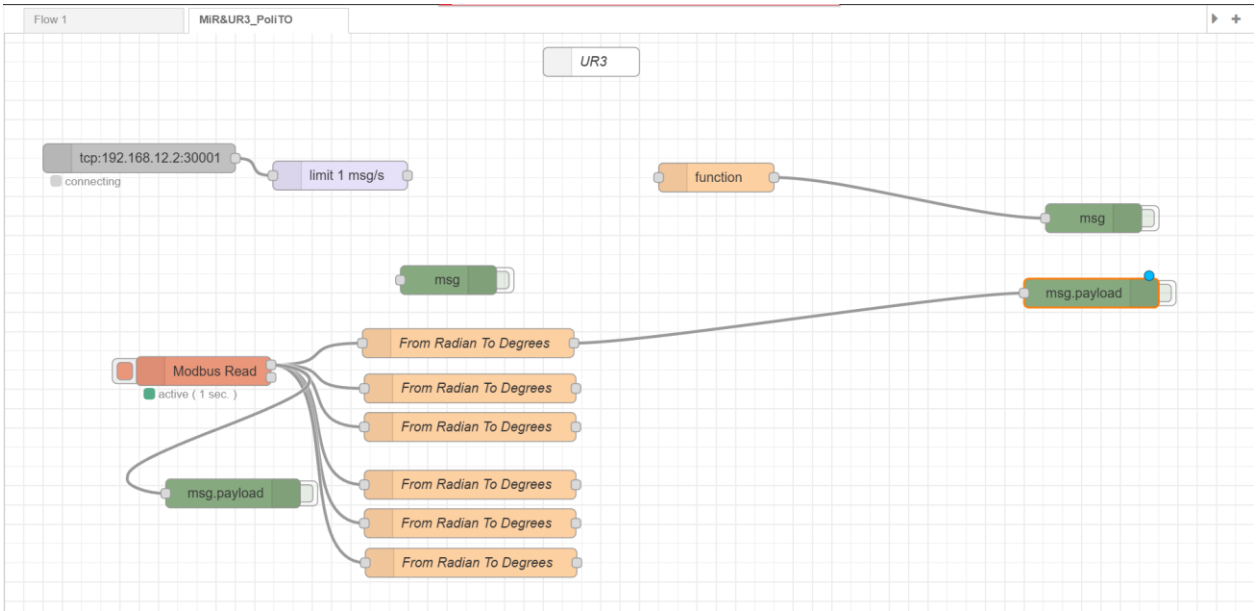
- This flow processes GPS data (X and Y coordinates) and yaw orientation from the mobile robot. Data is extracted using MODbus nodes and converted from binary buffer formats into floating-point values for further processing.
- The position and orientation data are then merged into a single JSON payload and published to MQTT topics such as posix, positionZ, and yaaw.
- The flow ensures that Unity receives accurate positional data, enabling real-time visualization of the mobile robot's movements.



MIR Mobile robot Data processing in NodeRed

3. **Flow for UR3 Robot Joint Data**:

- The UR3 robots transmit joint angle data via the Real-Time Data Exchange (RTDE) protocol. A Node-RED flow captures this data, converts joint angles from radians to degrees (to align with Unity's requirements), and publishes them to individual MQTT topics (ur3j1, ur3j2, etc.).
- This flow allows Unity to animate the UR3 robot with smooth and realistic joint movements.



UR3 NodeRed Data processing

Unity's Role in Consuming Node-RED Outputs

Unity consumes the processed data published by Node-RED through C# scripts, enabling the Digital Twin to represent the physical robots accurately. In Unity, the 'ConnectionMgr' script manages the MQTT connection and subscribes to the respective topics, updating static variables in real time. These variables are accessed by other Unity scripts in order to animate the robot models.

The angles of joints, for example, come into Unity via MQTT topics for the ABB robot: 'abbj1' to 'abbj6'. Similarly, each of these is applied to a corresponding GameObject in Unity. Each of the joints has its own script that updates the rotation of the GameObject based on data received. In the same way, position and yaw data from the mobile robot are utilized for its movement and orientation within the virtual environment.

6.3.2. C# Code for Data Reception

C# scripting, as implemented within Unity, forms an essential part of the Digital Twin system for receiving data in real-time from the MQTT broker. Unity acts as a visualization platform where these scripts process the data published by Node-RED and apply that data to robotic models for rendering and control. Such smooth data flow ensures the Digital Twin reflects the exact status of physical robots with preciseness and responsiveness.

The 'ConnectionMgr' script is responsible for the reception of data within Unity. It opens an MQTT connection, listens for topics, and handles messages. Each MQTT topic is a parameter for the physical robot: joint angles for the ABB robot or position information for the mobile robot. This information is parsed and then stored into static variables for access from other Unity scripts. For example, the following piece of code demonstrates how incoming messages are processed and routed to their respective variables:

```
private void Client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
{
    string topic = e.Topic;
    string message = System.Text.Encoding.UTF8.GetString(e.Message);

    switch (topic)
    {
        case "abbj1":
            J1 = float.Parse(message);
            break;
        case "posix":
            posx = float.Parse(message);
```

```
            break;
        }
    }
```

This structured approach allows Unity to handle several data streams in parallel, which enables it to process real-time updates from all connected robots and their parameters.

In addition to data reception, the Unity scripts are designed with robust error handling mechanisms to ensure reliable operation in case of unexpected issues. Data validation checks filter out invalid or out-of-range values. For example, in case of received data that is out of the predefined limits, it logs a warning and skips setting the value. It also includes reconnect mechanisms in order to automatically restore the MQTT connection after network disruptions. Debugging logs further enhance the reliability by logging every piece of data received and the possible errors.

The centralized architecture of the script ‘ConnectionMgr’ allows for a neat management of data in Unity. Unity avoids the unnecessary creation of connections to the MQTT broker by storing the incoming data using static variables; hence, other scripts running under Unity can have access to these variables for robot model manipulation. This portability is an assurance of the efficiency and flexibility that Unity C# scripting does for real-time data manipulation.

6.3.3. Real-Time Animation of Robots

One of the most impressive visual parts of a Digital Twin system is the real-time animation of robots. Digital Twin makes sure that, through the application of received data from MQTT topics on Unity's 3D models, the movement and states of the physical robots are accurately mirrored in the virtual environment. Every robotic component, from a joint to a position or orientation, is responsible for dedicated scripts that make detailed and synchronized animations possible.

In the case of the ABB robot, each joint has been animated in a way that the corresponding MQTT topic (e.g., ‘abbj1’ for Joint 1) is mapped onto a GameObject in Unity. The static variables inside the script ‘ConnectionMgr’ store real-time joint angles and are used to set the rotation of each joint via the following script:

```
void FixedUpdate()
{
    transform.localEulerAngles = new Vector3(
        0f, // Fixed X-axis rotation
        (float)((-1) * ConnectionMgr.J1), // Y-axis rotation
```

```
        0f // Fixed Z-axis rotation
    );
}
```

This setup ensures that the virtual ABB robot's movements perfectly act like its physical counterpart.

Positional and orientation data were used to control the movement and rotation of the mobile robot in Unity. The position and yaw are supplied from the ‘posix’ and ‘yaaw’ MQTT topics, respectively, which are applied to update the position and direction of the robot. The script used to perform this is depicted by the following:

```
transform.position = Vector3.MoveTowards(
    transform.position,
    targetObject.transform.position,
    moveSpeed * Time.deltaTime
);

Vector3 targetDirection = targetObject.transform.forward;
Vector3 newDirection = Vector3.RotateTowards(
    transform.forward,
    targetDirection,
    rotationSpeed * Time.deltaTime,
    0f
);
transform.rotation = Quaternion.LookRotation(newDirection);
```

These scripts provided views of smooth, realistic animations of the mobile robot, and maintained its physical behaviour properly in the virtual environment. Integrating real-time data with Unity's animation capabilities ensures that the physical and virtual systems are kept in sync. Updates to the states of robots happen with very minimal latency, thereby setting the Digital Twin for dynamic and interactive monitoring, control, and experimentation. The Digital Twin leverages the power of Unity's 3D rendering combined with C# scripting to perfectly blend accuracy, efficiency, and adaptability.

6.3.4. Synchronizing Physical and Digital Twins

Real-time data exchange between the physical robots and their virtual equivalents is ensured inside the Digital Twin. The Node-RED is used for pre-processing the data coming from robots and publishes it on MQTT topics. At the same time, Unity C# scripts subscribe to these MQTT topics for updates.

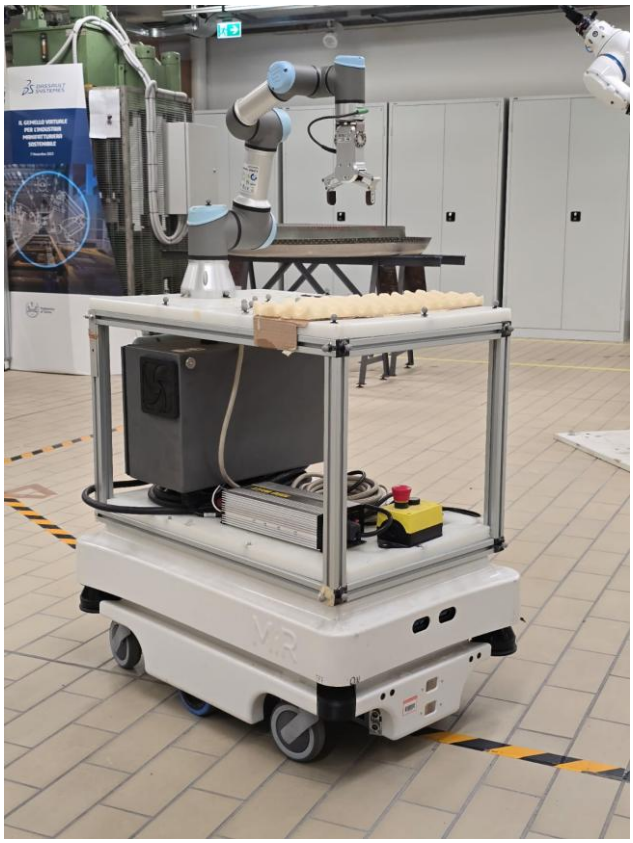
Key parameters include joint angles and positions, orientations that are continually streamed into Unity and applied to 3D models. This ensures that any changes that occur in the physical robots are instantaneously reflected within the virtual environment, allowing for a perfect connection to be maintained between the two systems.

Consistency and reliability in such cases are further ascertained with the use of error-handling mechanisms such as data validation and reconnection protocols. The obtained consistency of data provides the enabling framework for correct monitoring, predictive simulation, and interactive control of robotic systems using the Digital Twin.

7. Testing And Validation

Making sure the Digital Twin faithfully replicated the real-time motions of the physical robots was the main goal of the testing and validation procedure. Individual robot movements were seen in the Unity simulation and the actual lab setting to test the system. The accuracy of data collecting, MQTT transmission, and real-time synchronization in Unity were confirmed by independent testing of the ABB IRB 4400, UR3, and mobile robot. Following the successful completion of individual tests, all robots were merged to evaluate the Digital Twin's multi-robot coordination and interaction. The outcomes demonstrated dependable data flow, communication stability, and precise 3D display, confirming that the system successfully and without discernible delays duplicated the physical movements in Unity. These tests confirm that the built system works as planned for real-time monitoring and control, validating its efficacy.

ABB in the Lab



MIR Robot and UR3 in the Lab

7.1. KPIs to Be Considered for Testing and Future Improvements

The following Key Performance Indicators (KPIs) ought to be considered for testing and upcoming improvements in order to guarantee the dependability and effectiveness of the Digital Twin system:

1. **Real-Time Synchronization Accuracy:** calculates the amount of time that passes between a robot's actual movements and Unity's virtual representation of them.
2. **Data Transmission Reliability:** determines the proportion of successful, loss-free, and corrupted MQTT data packets received.
3. **Individual Robot Functionality:** makes certain that the Digital Twin accurately depicts the motions and behaviours of every robot.
4. **Multi-Robot Coordination:** evaluates how well several robots cooperate and synchronize in the virtual world.
5. **System Stability and Uptime:** evaluates the system's capacity to function continuously without crashing or experiencing unplanned malfunctions.
6. **Error Handling and Recovery:** evaluates the system's resilience to communication breakdowns, incorrect data, and network disconnections.
7. **Scalability and Expandability:** determines whether adding more robots or gadgets to the current Digital Twin framework is simple.

These KPIs are crucial standards for confirming system performance and pinpointing areas in need of further development.

8. Conclusion

The Digital Twin developed for multi-robot integration represents a huge leap in smart manufacturing and Industry 4.0. With the integration of advanced tools like Node-RED, MQTT, and Unity, the system has been able to successfully demonstrate its capabilities of mirroring the operations of physical robots in real time. The project focused on the development of a modular and scalable framework that would be able to handle various communication protocols and synchronize data for accurate 3D visualization.

The proposed solution proves that Digital Twins can play a leading role in innovative industrial environment monitoring, prediction, and interaction in real time. ABB, UR3, and mobile robots are included within one system framework, proving the flexibility of the approach. Among the achieved results are a robust acquisition of data, good efficiency in ensuring communication with the use of MQTT, and development of robust Unity scripts for animating and synchronizing the robotic models.

Despite these achievements, the project showed areas for improvement and future research as well. The growth of industrial processes in terms of complexity desperately calls for more intelligent and adaptive Digital Twin systems. Addressing these issues will be further work that enhances usability, scalability, and effectiveness for the Digital Twin.

8.1. Advantages of This Digital Twin Architecture

Numerous benefits that improve its functionality, scalability, and flexibility for industrial applications are provided by the developed Digital Twin (DT) architecture. The solution guarantees smooth multi-robot coordination, effective data handling, and real-time synchronization by incorporating contemporary technologies like Node-RED, MQTT, and Unity. The strengths of this architecture are highlighted by the following main advantages:

1. **Decoupling Services to Provide Flexibility and Modularity**
   Different components—data collection, processing, communication, and visualization—are separated from one another in this modular architecture.
   - This division makes it possible for: System components can be independently developed and modified without affecting the system's overall structure.
   - more freedom to upgrade components or integrate more robots.
   - Better maintainability since changes made to one service don't impact others.

2. **Effective Service for Data Provision**
   The Digital Twin aggregates real-time data from several robotic systems to serve as a centralized data centre.
   - The service that provides data guarantees: All linked robots are continuously monitored and receive real-time updates.
   - data flow that is consistent and structured, making analytics and visualization easier.
   - The ability to save historical data for analysis and preventative maintenance.

3. **Highly Customized and Open System**
   This architecture is notable for its open design and high degree of customization.
   Because the system is based on open-source technologies, it permits:
   - complete personalization of graphical components, communication protocols, and robot behaviour.
   - Simple adjustment to various robotic configurations and industrial settings.
   - a cooperative development process that makes it simple to apply future adjustments and improvements.

4. **Node-RED as an Independent and Scalable Platform**
   As a middleware, Node-RED makes it easier to integrate and communicate data.
   - Among its main benefits are:Data pipeline setup is made easier using Graphical Flow-Based Programming.
   - Scalability: Facilitates future system growth by enabling the integration of more robots and gadgets without requiring significant modification.
   - Error Handling and Debugging: Offers integrated debugging capabilities to track data flow in real time and identify connectivity problems.

5. **MQTT for Secure, Standardized, and Accessible Communication**
   The architecture offers several benefits by using MQTT as the primary communication protocol.
   - Security: Provides authentication and encryption for safe data transfer.
   - Widely used in industrial automation, this standard guaranteed interoperability with a range of robotic and Internet of Things technologies.

- Lightweight and Effective: It uses very little bandwidth and is therefore perfect for real-time applications.
- By using a publish-subscribe approach, subscription-based access enables data to be sent to various clients (such as monitoring systems or Unity) without requiring direct device-to-device interactions.

6. **AI and Predictive Maintenance Integration**

The architecture offers a starting point for future integration of machine learning (ML) and artificial intelligence (AI).

- This would enable: Predictive maintenance is the process of anticipating failures and scheduling maintenance before problems arise by using trends in past data.
- Anomaly Detection: AI-powered surveillance to identify anomalous robot activity and provide remedial measures.
- Optimization Algorithms: Enhancing robotic productivity through workflow optimization and movement pattern analysis.

7. **Real-Time Monitoring and Control**

The system makes use of Node-RED, MQTT, and Unity to provide interactive control and real-time robot monitoring.

- Instant Status Updates: Robot states can be seen by operators, who can react quickly to system modifications.
- Remote Access: By enabling monitoring and control from many locations, the system architecture enhances operational flexibility.
- Decreased Downtime: System failures and disruptions are minimized by prompt issue identification and resolution.

8. **Scalability for Future Expansion**

Because of its great scalability, the Digital Twin can expand to meet changing industrial demands.

- Supports Multiple Robots: Without completely redesigning the system, more robotic units can be added.
- Adaptability of Cloud and Edge Computing: Upcoming versions may employ edge computing to speed up reaction times or link to cloud platforms for remote processing.
- Flexible Data Sources: In addition to robots, the system can integrate environmental monitoring equipment, production line data, and Internet of Things sensors.

9. **Cost-Efficiency and Resource Optimization**

By providing a virtual representation of physical robots, the system reduces operational costs by:

- Minimizing Physical Testing: Virtual simulations help test robotic movements and workflows before implementing changes in real-world systems.
- Reducing Equipment Wear: Continuous monitoring and predictive maintenance reduce unnecessary strain on physical robots.
- Energy Efficiency: Optimized robot coordination and scheduling improve energy consumption.

10. **Enhanced Collaboration and Training**
    In the automation and robotics sectors, the Digital Twin facilitates improved workforce training and collaboration.
    - Operator Training Without Risk: New hires can receive training virtually without having an impact on actual operations.
    - Before deploying changes in a live system, engineers can use simulation to evaluate the effects of various approaches.
    - Cross-Team Collaboration: Without needing physical robot access, several teams (such as automation engineers and data analysts) can collaborate in the same space.

11. **Interoperability with Industrial Systems**
    The architecture is future-proof and flexible enough to accommodate new technologies because it facilitates integration with current industrial automation systems.
    - Can interface with SCADA, MES, and PLCs for smooth industrial automation.
    - Connects more smart devices to expand IoT capabilities for thorough system monitoring.
    - It is possible to integrate future AI and digital manufacturing applications without significantly altering the fundamental design.

8.2. Future Work

There are numerous suggestions for improving and growing the current system:

1. **Integration of AI and Machine Learning**: The integration of AI and machine learning skills is the next step in the development of the digital twin. Proactive maintenance is made possible using predictive analytics to spot possible problems before they arise. Additionally, robot cooperation can be improved by machine learning techniques, increasing productivity in multi-robot activities. AI may, for instance, anticipate bottlenecks in robot workflows and instantly provide different approaches.
2. **Extended Robot Network:** The Robot Network can be extended by scaling up the current implementation in managing a wider robotic network with much more diversity, integrating into one system autonomous drones, collaborative robots, and other industrial systems. Further work can be addressed to the problems of dealing with high-frequency data streams from a large number of sources and handling the synchronization issue.
3. **Improved User Interaction**: In the future, development of the Digital Twin could include designing a graphical interface for operators. Such an interface would intuitively enable users to view robot states, send commands, and visualize complex data trends. In addition, the integration of AR technologies will offer immersive interaction in which operators can view and control the Digital Twin directly within a physical workspace.
4. **Edge/Cloud Computing:** As cloud-based architectures are being taken up by the industrial system, moving the Digital Twin into the cloud would eventually enable

centralized control and storing of data. This again might facilitate collaboration across geographically distributed sites. Further exploration is foreseen for edge computing in processing data locally next to the robots to reduce latency and grant real-time performance if it's high demanding.

5. **Advanced Testing and Validation:** While functionality and synchronization have been tested for the system, there is a further need to conduct validation under industrial conditions. Stress testing of the system with higher complexity of tasks that will require collaboration by multiple robots will also reveal the system's limit and possible optimization. Such would be a simulation at a factory scale with various workloads, which shows points where the system may need more resources or optimization.
6. **Decision-making Digital Twin:** Beyond the role of visualization and monitoring, the Digital Twin can become a decision-making tool. The system would be able to simulate several scenarios and recommend the best course of action by combining real-time data with past performance indicators. When it comes to a production line, for instance, the Digital Twin might recommend a reconfiguration that considers both the present and the future to optimize throughput.

9. Reference

- *The digital twin of an industrial production line within the industry 4.0 concept. (2017, June 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/7976223*
- *Digital Twin in Industry 4.0: Technologies, applications and challenges. (2019, July 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/8972134*
- *Židek, K., Piteľ, J., Adámek, M., Lazorík, P., & Hošovský, A. (2020). Digital twin of experimental smart Manufacturing assembly System for Industry 4.0 concept. Sustainability, 12(9), 3658. https://doi.org/10.3390/su12093658*
- *A Methodology for Digital twin modeling and Deployment for Industry 4.0. (2021, April 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9247401*
- *Durão, L. F. C. S., Haag, S., Anderl, R., Schützer, K., & Zancul, E. (2018). Digital Twin requirements in the context of Industry 4.0. In IFIP advances in information and communication technology (pp. 204–214). https://doi.org/10.1007/978-3-030-01614-2_19*
- *Kenett, R. S., & Bortman, J. (2021). The digital twin in Industry 4.0: A wide-angle perspective. Quality and Reliability Engineering International, 38(3), 1357–1366. https://doi.org/10.1002/qre.2948*
- *Garg, G., Kuts, V., & Anbarjafari, G. (2021). Digital Twin for FANUC robots: industrial robot programming and simulation using virtual reality. Sustainability, 13(18), 10336. https://doi.org/10.3390/su131810336*
- *Erdei, T. I., Krakó, R., & Husi, G. (2022). Design of a digital twin training centre for an industrial robot arm. Applied Sciences, 12(17), 8862. https://doi.org/10.3390/app12178862*
- *Kuts, V., Cherezova, N., Sarkans, M., & Otto, T. (2020). Digital Twin: industrial robot kinematic model integration to the virtual reality environment. Journal of Machine Engineering, 20(2), 53–64. https://doi.org/10.36897/jme/120182*
- *Architecture for Digital Twin implementation focusing on Industry 4.0. (2020, May 1). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9082917*
- *Jacoby, M., & Usländer, T. (2020). Digital Twin and Internet of Things—Current Standards landscape. Applied Sciences, 10(18), 6519. https://doi.org/10.3390/app10186519*
- *Schnicke, F., Kuhn, T., & Antonino, P. O. (2020). Enabling industry 4.0 Service-Oriented architecture through digital twins. In Communications in computer and information science (pp. 490–503). https://doi.org/10.1007/978-3-030-59155-7_35*
- *Industrial IoT and Digital Twins for a Smart Factory: An open-source toolkit for application design and benchmarking. (2020, June 1). IEEE Conference Publication | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/9119497*
- *Human, C., Basson, A. H., & Kruger, K. (2021). Digital Twin Data Pipeline using MQTT in SLADTA. In Studies in computational intelligence (pp. 111–122). https://doi.org/10.1007/978-3-030-69373-2_7*

- *Bao, J., Guo, D., Li, J., & Zhang, J. (2018). The modelling and operations for the digital twin in the context of manufacturing. Enterprise Information Systems, 13(4), 534–556. https://doi.org/10.1080/17517575.2018.1526324*
- *C2PS: a digital twin architecture reference model for the Cloud-Based Cyber-Physical systems. (2017). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/7829368*
- *Thijssen, E. A. & Eindhoven University of Technology. (2021). MQTT based Communication Framework for AGVs in a Digital Twin [Thesis]. https://pure.tue.nl/ws/portalfiles/portal/168495991/0810786_E._A.Thijssen.pdf*
- *Balla, M., Haffner, O., Kučera, E., & Cigánek, J. (2023). Educational case studies: Creating a digital twin of the production line in TIA Portal, Unity, and Game4Automation Framework. Sensors, 23(10), 4977. https://doi.org/10.3390/s23104977*
- *Guerra-Zubiaga, D., Kuts, V., Mahmood, K., Bondar, A., Nasajpour-Esfahani, N., & Otto, T. (2021). An approach to develop a digital twin for industry 4.0 systems: manufacturing automation case studies. International Journal of Computer Integrated Manufacturing, 34(9), 933–949. https://doi.org/10.1080/0951192x.2021.1946857*
- *Vidal-Balea, A., Blanco-Novoa, O., Fraga-Lamas, P., Vilar-Montesinos, M., & Fernández-Caramés, T. M. (2022). A collaborative industrial augmented reality digital twin: Developing the future of Shipyard 4.0. In Springer eBooks (pp. 104–120). https://doi.org/10.1007/978-3-031-06371-8_8*
- *Gallala, A., Kumar, A. A., Hichri, B., & Plapper, P. (2022). Digital twin for Human–Robot interactions by means of industry 4.0 enabling technologies. Sensors, 22(13), 4950. https://doi.org/10.3390/s22134950*
- *Digital twin and big data towards smart manufacturing and industry 4.0: 360-degree comparison. (2018). IEEE Journals & Magazine | IEEE Xplore. https://ieeexplore.ieee.org/abstract/document/8258937*
- *Agostino, Í. R. S., Broda, E., Frazzon, E. M., & Freitag, M. (2020). Using a digital twin for production planning and control in industry 4.0. In International series in management science/operations research/International series in operations research & management science (pp. 39–60). https://doi.org/10.1007/978-3-030-43177-8_3*
- *Malykhina, G., & Tarkhov, D. (2018). Digital twin technology as a basis of the industry in future. ˜ the □European Proceedings of Social & Behavioural Sciences, 416–428. https://doi.org/10.15405/epsbs.2018.12.02.45*
- *Zhou, J., Zhang, S., & Gu, M. (2022). Revisiting digital twins: Origins, fundamentals, and practices. Frontiers of Engineering Management, 9(4), 668–676. https://doi.org/10.1007/s42524-022-0216-2*
- *Piromalis, D., & Kantaros, A. (2022). Digital Twins in the Automotive Industry: The Road toward Physical-Digital Convergence. Applied System Innovation, 5(4), 65. https://doi.org/10.3390/asi5040065*

10. Appendices

The appendices contain information that is supportive and gives further elaboration on what is contained in the body of the report. The section contains detailed technical information, code samples, and further diagrams that are too extensive for the primary sections but are essential to understand the implementation and methodologies used in this project. The appendices serve as a reference for readers who wish to explore the technical aspects of the Digital Twin system in greater depth.

10.1. Detailed C# Code

This appendix contains complete listings of the C# scripts used in Unity to integrate the Digital Twin system. Each script is presented with explanations of its purpose, functionality, and integration within the broader system.

Unity UI C# code for UR3:

```
using System;
using System.Text;
// Unity
using UnityEngine;
using UnityEngine.UI;
// TM
using TMPro;
public class main_ui_control : MonoBehaviour
{
    // -------------------- GameObject -------------------- //
    public GameObject camera_obj;
    // -------------------- Image -------------------- //
    public Image connection_panel_img, diagnostic_panel_img, joystick_panel_img;
    public Image connection_info_img;
    // -------------------- TMP_InputField -------------------- //
    public TMP_InputField ip_address_txt;
    // -------------------- Float -------------------- //
    private float ex_param = 100f;
    // -------------------- TextMeshProUGUI -------------------- //
    public TextMeshProUGUI position_x_txt, position_y_txt, position_z_txt;
    public TextMeshProUGUI position_rx_txt, position_ry_txt, position_rz_txt;
```

```
    public TextMeshProUGUI position_j1_txt, position_j2_txt, position_j3_txt;
    public TextMeshProUGUI position_j4_txt, position_j5_txt, position_j6_txt;
    public TextMeshProUGUI connectionInfo_txt;
    // -------------------- UTF8Encoding -------------------- //
    private UTF8Encoding utf8 = new UTF8Encoding();

    // ------------------------------------------------------------------------------------------------------------------ //
    // ------------------------------------------------ INITIALIZATION {START} ------------------------------------------------
- //
    // ------------------------------------------------------------------------------------------------------------------ //
    void Start()
    {
        // Connection information {image} -> Connect/Disconnect
        connection_info_img.GetComponent<Image>().color = new Color32(255, 0, 48, 50);
        // Connection information {text} -> Connect/Disconnect
        connectionInfo_txt.text = "Disconnect";

        // Panel Initialization -> Connection/Diagnostic/Joystick Panel
        connection_panel_img.transform.localPosition = new Vector3(1215f + (ex_param), 0f, 0f);
        diagnostic_panel_img.transform.localPosition = new Vector3(780f + (ex_param), 0f, 0f);
        joystick_panel_img.transform.localPosition = new Vector3(1550f + (ex_param), 0f, 0f);

        // Position {Cartesian} -> X..Z
        position_x_txt.text = "0.00";
        position_y_txt.text = "0.00";
        position_z_txt.text = "0.00";
        // Position {Rotation} -> EulerAngles(RX..RZ)
        position_rx_txt.text = "0.00";
        position_ry_txt.text = "0.00";
        position_rz_txt.text = "0.00";
        // Position Joint -> 1 - 6
        position_j1_txt.text = "0.00";
        position_j2_txt.text = "0.00";
        position_j3_txt.text = "0.00";
        position_j4_txt.text = "0.00";
```

```
        position_j5_txt.text = "0.00";
        position_j6_txt.text = "0.00";

        // Robot IP Address
        ip_address_txt.text = "127.0.0.1";

        // Auxiliary first command -> Write initialization position/rotation with acceleration/time to the robot controller
        // command (string value)
        ur_data_processing.UR_Control_Data.aux_command_str = "speedl([0.0,0.0,0.0,0.0,0.0,0.0], a = 0.15, t = 0.03)" + "\n";
        // get bytes from string command
        ur_data_processing.UR_Control_Data.command = utf8.GetBytes(ur_data_processing.UR_Control_Data.aux_command_str);
    }

    // ------------------------------------------------------------------------------------------------------------------- //
    // ------------------------------------------- MAIN FUNCTION {Cyclic} ------------------------------------------------ //
    // ------------------------------------------------------------------------------------------------------------------- //
    void FixedUpdate()
    {
        // Robot IP Address (Read) -> TCP/IP
        ur_data_processing.UR_Stream_Data.ip_address = ip_address_txt.text;
        // Robot IP Address (Write) -> TCP/IP
        ur_data_processing.UR_Control_Data.ip_address = ip_address_txt.text;

        // ------------------------ Connection Information ------------------------//
        // If the button (connect/disconnect) is pressed, change the color and text
        if (ur_data_processing.GlobalVariables_Main_Control.connect == true)
        {
            // green color
            connection_info_img.GetComponent<Image>().color = new Color32(135, 255, 0, 50);
            connectionInfo_txt.text = "Connect";
        }
        else if(ur_data_processing.GlobalVariables_Main_Control.disconnect == true)
```

```
        {
            // red color
            connection_info_img.GetComponent<Image>().color = new Color32(255, 0, 48, 50);
            connectionInfo_txt.text = "Disconnect";
        }

        // ------------------------ Cyclic read parameters {diagnostic panel} ------------------------ //
        // Position {Cartesian} -> X..Z
        position_x_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.C_Position[0] * (1000f), 2)).ToString();
        position_y_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.C_Position[1] * (1000f), 2)).ToString();
        position_z_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.C_Position[2] * (1000f), 2)).ToString();
        // Position {Rotation} -> EulerAngles(RX..RZ)
        position_rx_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.C_Orientation[0] * (180 / Math.PI), 2)).ToString();
        position_ry_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.C_Orientation[1] * (180 / Math.PI), 2)).ToString();
        position_rz_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.C_Orientation[2] * (180 / Math.PI), 2)).ToString();
        // Position Joint -> 1 - 6
        position_j1_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.J_Orientation[0] * (180 / Math.PI), 2)).ToString();
        position_j2_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.J_Orientation[1] * (180 / Math.PI), 2)).ToString();
        position_j3_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.J_Orientation[2] * (180 / Math.PI), 2)).ToString();
        position_j4_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.J_Orientation[3] * (180 / Math.PI), 2)).ToString();
        position_j5_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.J_Orientation[4] * (180 / Math.PI), 2)).ToString();
        position_j6_txt.text = ((float)Math.Round(ur_data_processing.UR_Stream_Data.J_Orientation[5] * (180 / Math.PI), 2)).ToString();
    }

    // ------------------------------------------------------------------------------------------------------------ //
    // ------------------------------------------------- FUNCTIONS ------------------------------------------------- //
    // ------------------------------------------------------------------------------------------------------------ //
```

```
// -------------------- Destroy Blocks -------------------- //
void OnApplicationQuit()
{
    // Destroy all
    Destroy(this);
}

// -------------------- Connection Panel -> Visible On -------------------- //
public void TaskOnClick_ConnectionBTN()
{
    // visible on
    connection_panel_img.transform.localPosition = new Vector3(0f, 0f, 0f);
    // visible off
    diagnostic_panel_img.transform.localPosition = new Vector3(780f + (ex_param), 0f, 0f);
    joystick_panel_img.transform.localPosition = new Vector3(1550f + (ex_param), 0f, 0f);
}

// -------------------- Connection Panel -> Visible off -------------------- //
public void TaskOnClick_EndConnectionBTN()
{
    connection_panel_img.transform.localPosition = new Vector3(1215f + (ex_param), 0f, 0f);
}

// -------------------- Diagnostic Panel -> Visible On -------------------- //
public void TaskOnClick_DiagnosticBTN()
{
    // visible on
    diagnostic_panel_img.transform.localPosition = new Vector3(0f, 0f, 0f);
    // visible off
    connection_panel_img.transform.localPosition = new Vector3(1215f + (ex_param), 0f, 0f);
    joystick_panel_img.transform.localPosition = new Vector3(1550f + (ex_param), 0f, 0f);
}
```

```
// ---------- Diagnostic Panel -> Visible Off ---------- //
public void TaskOnClick_EndDiagnosticBTN()
{
    diagnostic_panel_img.transform.localPosition = new Vector3(780f + (ex_param), 0f, 0f);
}

// ---------- Joystick Panel -> Visible On ---------- //
public void TaskOnClick_JoystickBTN()
{
    // visible on
    joystick_panel_img.transform.localPosition = new Vector3(-265f, -129f, 0f);
    // visible off
    connection_panel_img.transform.localPosition = new Vector3(1215f + (ex_param), 0f, 0f);
    diagnostic_panel_img.transform.localPosition = new Vector3(780f + (ex_param), 0f, 0f);
}

// ---------- Joystick Panel -> Visible Off ---------- //
public void TaskOnClick_EndJoystickBTN()
{
    joystick_panel_img.transform.localPosition = new Vector3(1550f + (ex_param), 0f, 0f);
}

// ---------- Camera Position -> Right ---------- //
public void TaskOnClick_CamViewRBTN()
{
    camera_obj.transform.localPosition    = new Vector3(0.114f, 2.64f, -2.564f);
    camera_obj.transform.localEulerAngles = new Vector3(10f, -30f, 0f);
}

// ---------- Camera Position -> Left ---------- //
public void TaskOnClick_CamViewLBTN()
{
    camera_obj.transform.localPosition = new Vector3(-3.114f, 2.64f, -2.564f);
    camera_obj.transform.localEulerAngles = new Vector3(10f, 30f, 0f);
```

```
    }

    // -------------------- Camera Position -> Home (in front) -------------------- //
    public void TaskOnClick_CamViewHBTN()
    {
        camera_obj.transform.localPosition = new Vector3(-1.5f, 2.2f, -3.5f);
        camera_obj.transform.localEulerAngles = new Vector3(0f, 0f, 0f);
    }

    // -------------------- Camera Position -> Top -------------------- //
    public void TaskOnClick_CamViewTBTN()
    {
        camera_obj.transform.localPosition = new Vector3(-1.2f, 4f, 0f);
        camera_obj.transform.localEulerAngles = new Vector3(90f, 0f, 0f);
    }

    // -------------------- Connect Button -> is pressed -------------------- //
    public void TaskOnClick_ConnectBTN()
    {
        ur_data_processing.GlobalVariables_Main_Control.connect    = true;
        ur_data_processing.GlobalVariables_Main_Control.disconnect = false;
    }

    // -------------------- Disconnect Button -> is pressed -------------------- //
    public void TaskOnClick_DisconnectBTN()
    {
        ur_data_processing.GlobalVariables_Main_Control.connect    = false;
        ur_data_processing.GlobalVariables_Main_Control.disconnect = true;
    }

}
```

Unity MQTT Client handler “M2MqttUnityClient”:

```
using System;
```

```
using System.Collections;
using System.Collections.Generic;
using UnityEngine;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

/// <summary>
/// Adaptation for Unity of the M2MQTT library (https://github.com/eclipse/paho.mqtt.m2mqtt),
/// modified to run on UWP (also tested on Microsoft HoloLens).
/// </summary>
namespace M2MqttUnity
{
    /// <summary>
    /// Generic MonoBehavior wrapping a MQTT client, using a double buffer to postpone message processing in the main thread.
    /// </summary>
    public class M2MqttUnityClient : MonoBehaviour
    {
        [Header("MQTT broker configuration")]
        [Tooltip("IP address or URL of the host running the broker")]
        public string brokerAddress = "localhost";
        [Tooltip("Port where the broker accepts connections")]
        public int brokerPort = 1883;
        [Tooltip("Use encrypted connection")]
        public bool isEncrypted = false;
        [Header("Connection parameters")]
        [Tooltip("Connection to the broker is delayed by the the given milliseconds")]
        public int connectionDelay = 500;
        [Tooltip("Connection timeout in milliseconds")]
        public int timeoutOnConnection = MqttSettings.MQTT_CONNECT_TIMEOUT;
        [Tooltip("Connect on startup")]
        public bool autoConnect = false;
        [Tooltip("UserName for the MQTT broker. Keep blank if no user name is required.")]
        public string mqttUserName = null;
        [Tooltip("Password for the MQTT broker. Keep blank if no password is required.")]
```

```
public string mqttPassword = null;

/// <summary>
/// Wrapped MQTT client
/// </summary>
protected MqttClient client;

private List<MqttMsgPublishEventArgs> messageQueue1 = new List<MqttMsgPublishEventArgs>();
private List<MqttMsgPublishEventArgs> messageQueue2 = new List<MqttMsgPublishEventArgs>();
private List<MqttMsgPublishEventArgs> frontMessageQueue = null;
private List<MqttMsgPublishEventArgs> backMessageQueue = null;
private bool mqttClientConnectionClosed = false;
private bool mqttClientConnected = false;

/// <summary>
/// Event fired when a connection is successfully established
/// </summary>
public event Action ConnectionSucceeded;
/// <summary>
/// Event fired when failing to connect
/// </summary>
public event Action ConnectionFailed;

/// <summary>
/// Connect to the broker using current settings.
/// </summary>
public virtual void Connect()
{
    if (client == null || !client.IsConnected)
    {
        StartCoroutine(DoConnect());
    }
}
```

```
/// <summary>
/// Disconnect from the broker, if connected.
/// </summary>
public virtual void Disconnect()
{
    if (client != null)
    {
        StartCoroutine(DoDisconnect());
    }
}

/// <summary>
/// Override this method to take some actions before connection (e.g. display a message)
/// </summary>
protected virtual void OnConnecting()
{
    Debug.LogFormat("Connecting to broker on {0}:{1}...\n", brokerAddress, brokerPort.ToString());
}

/// <summary>
/// Override this method to take some actions if the connection succeeded.
/// </summary>
protected virtual void OnConnected()
{
    Debug.LogFormat("Connected to {0}:{1}...\n", brokerAddress, brokerPort.ToString());

    SubscribeTopics();

    if (ConnectionSucceeded != null)
    {
        ConnectionSucceeded();
    }
}
```

```
    /// <summary>
    /// Override this method to take some actions if the connection failed.
    /// </summary>
    protected virtual void OnConnectionFailed(string errorMessage)
    {
        Debug.LogWarning("Connection failed.");
        if (ConnectionFailed != null)
        {
            ConnectionFailed();
        }
    }

    /// <summary>
    /// Override this method to subscribe to MQTT topics.
    /// </summary>
    protected virtual void SubscribeTopics()
    {
    }

    /// <summary>
    /// Override this method to unsubscribe to MQTT topics (they should be the same you subscribed to with SubscribeTopics() ).
    /// </summary>
    protected virtual void UnsubscribeTopics()
    {
    }

    /// <summary>
    /// Disconnect before the application quits.
    /// </summary>
    protected virtual void OnApplicationQuit()
    {
        CloseConnection();
    }
```

```
/// <summary>
/// Initialize MQTT message queue
/// Remember to call base.Awake() if you override this method.
/// </summary>
protected virtual void Awake()
{
    frontMessageQueue = messageQueue1;
    backMessageQueue = messageQueue2;
}

/// <summary>
/// Connect on startup if autoConnect is set to true.
/// </summary>
protected virtual void Start()
{
    if (autoConnect)
    {
        Connect();
    }
}

/// <summary>
/// Override this method for each received message you need to process.
/// </summary>
protected virtual void DecodeMessage(string topic, byte[] message)
{
    Debug.LogFormat("Message received on topic: {0}", topic);
}

/// <summary>
/// Override this method to take some actions when disconnected.
/// </summary>
protected virtual void OnDisconnected()
{
```

```
    Debug.Log("Disconnected.");
}

/// <summary>
/// Override this method to take some actions when the connection is closed.
/// </summary>
protected virtual void OnConnectionLost()
{
    Debug.LogWarning("CONNECTION LOST!");
}

/// <summary>
/// Processing of income messages and events is postponed here in the main thread.
/// Remember to call ProcessMqttEvents() in Update() method if you override it.
/// </summary>
protected virtual void Update()
{
    ProcessMqttEvents();
}

protected virtual void ProcessMqttEvents()
{
    // process messages in the main queue
    SwapMqttMessageQueues();
    ProcessMqttMessageBackgroundQueue();
    // process messages income in the meanwhile
    SwapMqttMessageQueues();
    ProcessMqttMessageBackgroundQueue();

    if (mqttClientConnectionClosed)
    {
        mqttClientConnectionClosed = false;
        OnConnectionLost();
    }
```

```
    }

    private void ProcessMqttMessageBackgroundQueue()
    {
        foreach (MqttMsgPublishEventArgs msg in backMessageQueue)
        {
            DecodeMessage(msg.Topic, msg.Message);
        }
        backMessageQueue.Clear();
    }

    /// <summary>
    /// Swap the message queues to continue receiving message when processing a queue.
    /// </summary>
    private void SwapMqttMessageQueues()
    {
        frontMessageQueue = frontMessageQueue == messageQueue1 ? messageQueue2 : messageQueue1;
        backMessageQueue = backMessageQueue == messageQueue1 ? messageQueue2 : messageQueue1;
    }

    private void OnMqttMessageReceived(object sender, MqttMsgPublishEventArgs msg)
    {
        frontMessageQueue.Add(msg);
    }

    private void OnMqttConnectionClosed(object sender, EventArgs e)
    {
        // Set unexpected connection closed only if connected (avoid event handling in case of controlled
disconnection)
        mqttClientConnectionClosed = mqttClientConnected;
        mqttClientConnected = false;
    }

    /// <summary>
    /// Connects to the broker using the current settings.
```

```
    /// </summary>
    /// <returns>The execution is done in a coroutine.</returns>
    private IEnumerator DoConnect()
    {
        // wait for the given delay
        yield return new WaitForSecondsRealtime(connectionDelay / 1000f);
        // leave some time to Unity to refresh the UI
        yield return new WaitForEndOfFrame();

        // create client instance
        if (client == null)
        {
            try
            {
#if (!UNITY_EDITOR && UNITY_WSA_10_0 && !ENABLE_IL2CPP)
                client = new MqttClient(brokerAddress,brokerPort,isEncrypted, isEncrypted ? MqttSslProtocols.SSLv3 : MqttSslProtocols.None);
#else
                client = new MqttClient(brokerAddress, brokerPort, isEncrypted, null, null, isEncrypted ? MqttSslProtocols.SSLv3 : MqttSslProtocols.None);
                //System.Security.Cryptography.X509Certificates.X509Certificate cert = new System.Security.Cryptography.X509Certificates.X509Certificate();
                //client = new MqttClient(brokerAddress, brokerPort, isEncrypted, cert, null, MqttSslProtocols.TLSv1_0, MyRemoteCertificateValidationCallback);
#endif
            }
            catch (Exception e)
            {
                client = null;
                Debug.LogErrorFormat("CONNECTION FAILED! {0}", e.ToString());
                OnConnectionFailed(e.Message);
                yield break;
            }
        }
        else if (client.IsConnected)
        {
```

```
            yield break;
        }
        OnConnecting();

        // leave some time to Unity to refresh the UI
        yield return new WaitForEndOfFrame();
        yield return new WaitForEndOfFrame();

        client.Settings.TimeoutOnConnection = timeoutOnConnection;
        string clientId = Guid.NewGuid().ToString();
        try
        {
            client.Connect(clientId, mqttUserName, mqttPassword);
        }
        catch (Exception e)
        {
            client = null;
            Debug.LogErrorFormat("Failed to connect to {0}:{1}\n (check client parameters: encryption,
address/port, username/password):\n{2}", brokerAddress, brokerPort, e.ToString());
            OnConnectionFailed(e.Message);
            yield break;
        }
        if (client.IsConnected)
        {
            client.ConnectionClosed += OnMqttConnectionClosed;
            // register to message received
            client.MqttMsgPublishReceived += OnMqttMessageReceived;
            mqttClientConnected = true;
            OnConnected();
        }
        else
        {
            OnConnectionFailed("CONNECTION FAILED!");
        }
    }
```

```
    private IEnumerator DoDisconnect()
    {
        yield return new WaitForEndOfFrame();
        CloseConnection();
        OnDisconnected();
    }

    private void CloseConnection()
    {
        mqttClientConnected = false;
        if (client != null)
        {
            if (client.IsConnected)
            {
                UnsubscribeTopics();
                client.Disconnect();
            }
            client.MqttMsgPublishReceived -= OnMqttMessageReceived;
            client.ConnectionClosed -= OnMqttConnectionClosed;
            client = null;
        }
    }

#if ((!UNITY_EDITOR && UNITY_WSA_10_0))
    private void OnApplicationFocus(bool focus)
    {
        // On UWP 10 (HoloLens) we cannot tell whether the application actually got closed or just minimized.
        // (https://forum.unity.com/threads/onapplicationquit-and-ondestroy-are-not-called-on-uwp-10.462597/)
        if (focus)
        {
            Connect();
        }
        else
```

```
        {
            CloseConnection();
        }
    }
#endif
  }
}
```

Unity ABB MQTT log C# code “AbbAloqa”:

```
using UnityEngine;
using System;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

public class AbbAloqa : MonoBehaviour
{
  public static float abbjoint1;
  public static float abbjoint2;
  public static float abbjoint3;
  public static float abbjoint4;
  public static float abbjoint5;
  public static float abbjoint6;

  private MqttClient client;
  public string brokerAddress = "lepiot.polito.it";
  public int brokerPort = 8081;

  public string[] topics = { "abb.j1", "abb.j2", "abb.j3", "abb.j4", "abb.j5", "abb.j6" };

  void Start()
  {
    client = new MqttClient(brokerAddress, brokerPort, false, null);
    client.MqttMsgPublishReceived += Client_MqttMsgPublishReceived;
    client.Connect(Guid.NewGuid().ToString());
```

```
    foreach (string topic in topics)
    {
        client.Subscribe(new string[] { topic }, new byte[] { MqttMsgBase.QOS_LEVEL_AT_LEAST_ONCE });
    }
}

private void Client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
{
    string topic = e.Topic;
    string message = System.Text.Encoding.UTF8.GetString(e.Message);

    Debug.Log($"Received message on {topic}: {message}");

    switch (topic)
    {
        case "abb.j1":
            abbjoint1 = float.Parse(message);
            break;
        case "abb.j2":
            abbjoint2 = float.Parse(message);
            break;
        case "abb.j3":
            abbjoint3 = float.Parse(message);
            break;
        case "abb.j4":
            abbjoint4 = float.Parse(message);
            break;
        case "abb.j5":
            abbjoint5 = float.Parse(message);
            break;
        case "abb.j6":
            abbjoint6 = float.Parse(message);
            break;
```

```
            default:
                Debug.LogWarning("Received message on an unrecognized topic: " + topic);
                break;
        }
    }

    void OnDestroy()
    {
        if (client != null && client.IsConnected)
        {
            client.Disconnect();
        }
    }
}
```

Unity MQTT Broker settings “BrokerSettings”:

```
using System;
using System.Xml.Serialization;
using UnityEngine;

namespace M2MqttUnity
{
    /// <summary>
    /// Serializable settings for MQTT broker configuration.
    /// </summary>
    [Serializable]
    [XmlType(TypeName = "broker-settings")]
    public class BrokerSettings
    {
        [Tooltip("Address of the host running the broker")]
        public string host = "localhost";

        [Tooltip("Port used to access the broker")]
        public int port = 1883;
```

```
        [Tooltip("Encrypted access to the broker")]
        public bool encrypted = false;

        [Tooltip("Optional alternate addresses, used if the previous host is not accessible")]
        public string[] alternateAddress;
    }
}
```

MQTT-Based Data Acquisition for ABB Robot and Mobile Robot Positioning in Unity C# code “ConnectIn_ABB”:

```
using UnityEngine;
using System;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

public class ConnectIn : MonoBehaviour
{
    public static float J1;
    public static float J2;
    public static float J3;
    public static float J4;
    public static float J5;
    public static float J6;
    public static float posx;
    public static float posZ;
    public static float yaw;

    private MqttClient client;
    public string brokerAddress = "lepiot.polito.it";
    public int brokerPort = 8081;

    public string[] topics = { "abb.j1", "abb.j2", "abb.j3", "abb.j4", "abb.j5", "abb.j6", "posix", "positionZ",
"yaaw" };
```

```
void Start()
{
    client = new MqttClient(brokerAddress, brokerPort, false, null);
    client.MqttMsgPublishReceived += Client_MqttMsgPublishReceived;
    client.Connect(Guid.NewGuid().ToString());

    foreach (string topic in topics)
    {
        client.Subscribe(new string[] { topic }, new byte[] { MqttMsgBase.QOS_LEVEL_AT_LEAST_ONCE });
    }
}

private void Client_MqttMsgPublishReceived(object sender, MqttMsgPublishEventArgs e)
{
    string topic = e.Topic;
    string message = System.Text.Encoding.UTF8.GetString(e.Message);

    //Debug.Log($"Received message on {topic}: {message}");

    switch (topic)
    {
        case "abb.j1":
            J1 = float.Parse(message);
            break;
        case "abb.j2":
            J2 = float.Parse(message);
            break;
        case "abb.j3":
            J3 = float.Parse(message);
            break;
        case "abb.j4":
            J4 = float.Parse(message);
            break;
        case "abb.j5":
```

```
            J5 = float.Parse(message);
            break;
        case "abb.j6":
            J6 = float.Parse(message);
            break;
        case "posix":
            posx = float.Parse(message);
            break;
        case "positionZ":
            posZ = float.Parse(message);
            break;
        case "yaaw":
            yaw = float.Parse(message);
            break;
        default:
            Debug.LogWarning("Received message on an unrecognized topic: " + topic);
            break;
      }
   }
   void OnDestroy()
   {
      if (client != null && client.IsConnected)
      {
         client.Disconnect();
      }
   }
}
```

Unity ABB Joint control “ABBJoint1”:

```
// System
using System;
//using System.Diagnostics;
// Unity
using UnityEngine;
```

```
using static UnityEngine.GraphicsBuffer;
//using Debug = UnityEngine.Debug;

public class ABBJoint1 : MonoBehaviour
{
    private Vector3 velocity = Vector3.zero; // Declare this as a class veriable
    public float smoothTime = 0.3f; //Adjust this value based on your needs

    void FixedUpdate()
    {
        try
        {
            Vector3 currentAngles = transform.localEulerAngles;
            Vector3 targetAngles = new Vector3(0f, (float)((1) * AbbAloqa.abbjoint1), 0f); //set your target euler
angle
            Vector3 smoothAngles = Vector3.SmoothDamp(currentAngles, targetAngles, ref velocity, smoothTime);
            transform.localEulerAngles = smoothAngles;

            //transform.localEulerAngles = new Vector3(0f, (float)((-1) * AbbAloqa.abbjoint1), 0f);
            Debug.Log("Received message1: " + AbbAloqa.abbjoint1);
            Debug.Log("Received message1: " + AbbAloqa.abbjoint2);
            Debug.Log("Received message1: " + AbbAloqa.abbjoint3);
            Debug.Log("Received message1: " + AbbAloqa.abbjoint4);
            Debug.Log("Received message1: " + AbbAloqa.abbjoint5);
            Debug.Log("Received message1: " + AbbAloqa.abbjoint6);
        }
        catch (Exception e)
        {
            Debug.Log("Exception:" + e);
        }
    }
    void OnApplicationQuit()
    {
        Destroy(this);
    }
```

```
}
```

Unity MIR robot position control “mir100trnsfrm”:

```
using System;
// Unity
using UnityEngine;

using Debug = UnityEngine.Debug;

public class mir100trnsfrm : MonoBehaviour
{

    //public float ZPosition = ConnectIn.posZ;

    void FixedUpdate()
    {
        try
        {
            // Update the game object's position in the x and y coordinates
            transform.localPosition = new Vector3(
                ConnectIn.posx, // X position
                transform.localPosition.y,
                ConnectIn.posZ // Y position

            );

            // Update the game object's rotation around the Z-axis (yaw angle)
            transform.localEulerAngles = new Vector3(
                -90f, // X rotation
                (float)((-1) * ConnectIn.yaw),
                0f // Y rotation
                 // Z rotation (yaw)
            );
```

```
            Debug.Log($"Received message on posX: {ConnectIn.posx}");
            Debug.Log($"Received message posZ : {ConnectIn.posZ}");
            Debug.Log($"Received message Yaw : {ConnectIn.yaw}");
        }
        catch (Exception e)
        {
            Debug.Log("Exception:" + e);
        }
    }

    void OnApplicationQuit()
    {
        Destroy(this);
    }
}
```