ICAS 2020 SHANGHAI
International Council of the Aeronautical Sciences
32nd Congress of the International Council of the Aeronautical Sciences
September 6-10, 2021
Pudong Shangri-La, Shanghai, China

# USAGE OF MQTT, ROS, AND AWS IN THE MANUFACTURING PROCESS OF AIRCRAFTS

L. S. Luna[1], G. A. P. Caurin[1] & M. L. Tronco[1]

[1]University of São Paulo, São Carlos/SP, Brazil

### Abstract

Recent advancements in technology promise to change many aspects of the industry. The proposal of Industry 4.0 is to integrate computational technologies to the factory, aiming to create distributed and autonomous cyber-physical systems. Systems that implement these technologies have great potential to increase security, control, and reduce costs in the supply chain. In this work, we study the usage of a system that uses MQTT, a common protocol in Industry 4.0 applications, and the framework ROS, common on robotics systems, to create a distributed system for monitoring and controlling the milling of aircraft body panels, and related processes. The application also uses tools provided by Amazon Web Services (AWS), to make it able to have cloud applications and to facilitate network connections between the devices.

**Keywords:** Industry 4.0, MQTT, Manufacturing, ROS, AWS

## 1. Introduction

Since it was first promoted, there have been many attempts to move the industry towards the enunciated Industry 4.0 goals. But even with increased inter-connectivity and many available technologies, many of those attempts still fall short [1]. In this work, we aimed to use some of the available tools that are already present in many sectors, and to describe an use case that integrates them in a context of aircraft manufacturing. We use the MQTT (Message Queueing Telemetry Transport) protocol, that has become industry standard because of its simplicity and efficiency [2]. It will be connected to a ROS (Robot Operating System) framework that will be able to control an manipulator robot, that is integrated in the manufacturing process of aircraft body panels. This connection will happen in a remote machine be handled through an AWS EC2 remote machine, which will also be used for data storage.

## 2. Materials

### 2.1 MQTT

The usage of the MQTT protocol is commonplace in the industry [2]. It is a application layer protocol designed for resource-constrained devices. Its messaging system works on a publisher-subscriber basis. It is a reliable system, having a TCP-IP transport layer with up to 2 levels of verification of message delivery. Yet, it presents a lower overhead than other protocols that rely on TCP-IP, like HTTP. [3] Its standard of transmitting all information through *json* strings encoded in UTF-8 allows a decreased size in the packages transmitted, as can be seen in Figure 1. These aspects make MQTT a bandwidth-efficient and energy-efficient protocol [4].

| bit | 7 | 6 | 5 | 4 | 3 | 2 | 1 | 0 |
|---|---|---|---|---|---|---|---|---|
| byte 1 | Message Type | | | | DUP Flag | QoS Level | | Ret |
| byte 2 | Remaining Length | | | | | | | |
| Variable Header | | | | | | | | |
| Payload | | | | | | | | |

Figure 1 – Example of MQTT message header (Source: [5])

### *2.1.1 Publisher-subscriber structure*

A publisher-subscriber messaging structure is a form of organization of protocols that reduces the amount of information that needs to be transmitted. There is a central broker that controls the roles of all the nodes involved. If a node has an information that needs to publish, it communicates with the broker and registers as a publisher to the topics which it wants to transmit its messages. Other nodes also communicate to the broker if they wish to subscribe to the topic, and therefore receive all messages that are published to that topic. Figure 2 illustrates what was described.

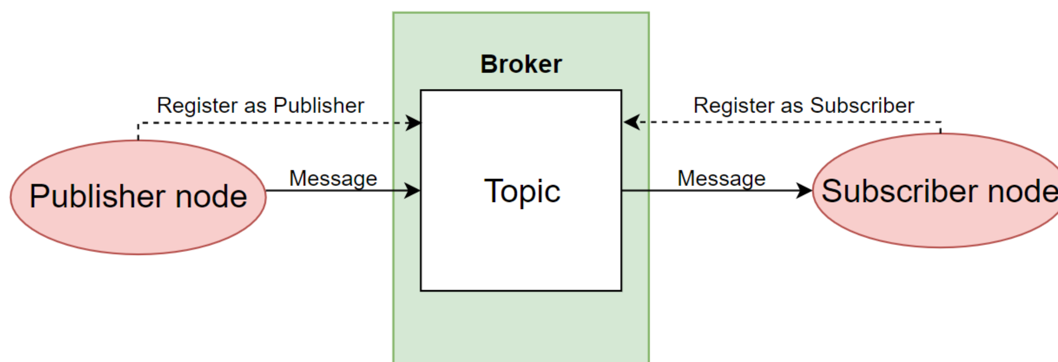


Figure 2 – Demonstration of a publisher-subscriber system

### *2.1.2 Quality of Service*

The MQTT protocol makes a distinction between three levels of quality of service (QoS) of message delivery [6]. The message flow of the different levels can be seen in Figure 3. In QoS 0, the messages will be delivered at most once. The client publishes the message to the broker and receives no acknowledgement that the message was delivered. There can be message loss, since the reliability level is only that of the TCP protocol operating underneath the MQTT protocol. This QoS is most appropriate for supervisor sensor data where the loss of single readings does not harm the long-time monitoring of the process.

In QoS 1, the messages will be delivered at least once. The client publishes its message and awaits for the acknowledgement of the broker. If the client receives no confirmation of message arrival, it publishes the message again with a DUP flag, denoting that the message is duplicated. At this QoS level, the DUP flag is not processed by the broker. When the broker receives a message it sends back a PUBACK packet with the published message identifier as confirmation. When the client receives the PUBACK, it ends the process. If a PUBACK message was lost, it is possible that the broker received the message more than once.

In QoS 2, the message will be delivered exactly once. This process works by a four-way handshake. The client publishes a message. The broker replies with a PUBREC packet marked with the message identifier. If the client receives no acknowledgement, it publishes the message again with an DUP flag. Once the client receives the PUBREC packet, it discards the original message, stores the PUBREC message and sends a PUBREL packet to the broker. When the broker receives this messages, it responds with a PUBCOMP packet to end the process. While the process is not finalized, the broker is storing the message identifier to assure that no message is published twice. When the process ends, both sides are the message was delivered and that the client has knowledge of that. This level is important in systems where duplicate messages could lead to problems, like billing systems, for example.

In this work, we mainly work with an QoS 0. The loss of packages should not be a worry with the intended usage. In future applications, implementing a control with sensor data in the side of the administrative network, the usage of higher levels of QoS might be needed.

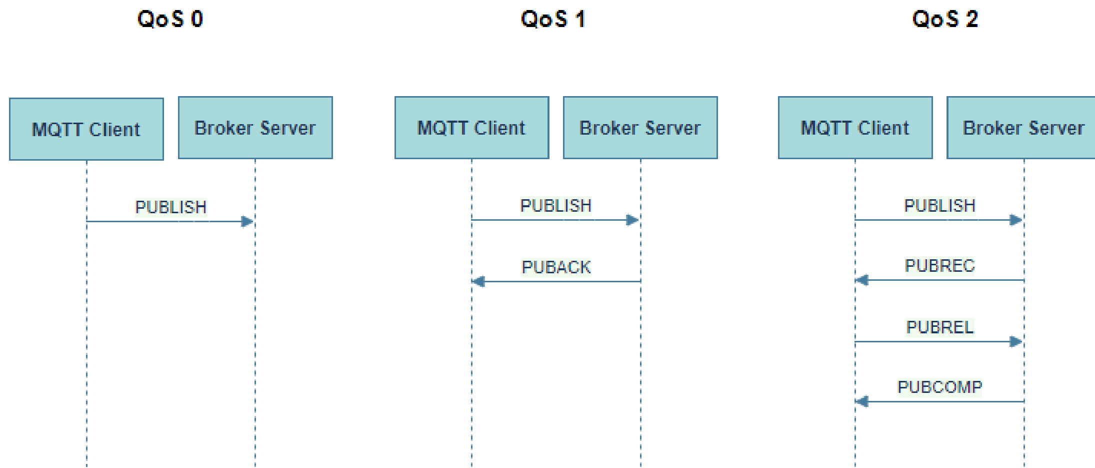


Figure 3 – Message flow of the different QoS levels

### *2.1.3 Mosquitto*

Mosquitto is a message broker that implements MQTT server and clients for use in light-weight applications [4]. It is suitable for use in micro controllers and low power devices. It provides simple publisher and subscriber command line functions for the MQTT protocol. Mosquitto is part of the Eclipse Foundation [7].

## 2.2 Node-RED

Node-RED is a light-weight application that provides an easy-to-use programming interface directly in the browser [8]. It is built on node.js, Therefore, its usage of json makes it compatible with protocols like MQTT. Node-RED provides a simple block-based programming suite that facilitates its use. It is also possible to use extensions that create simple interfaces that become available in the local network of the system. Figure 4 shows an example of the usage of Node-RED.

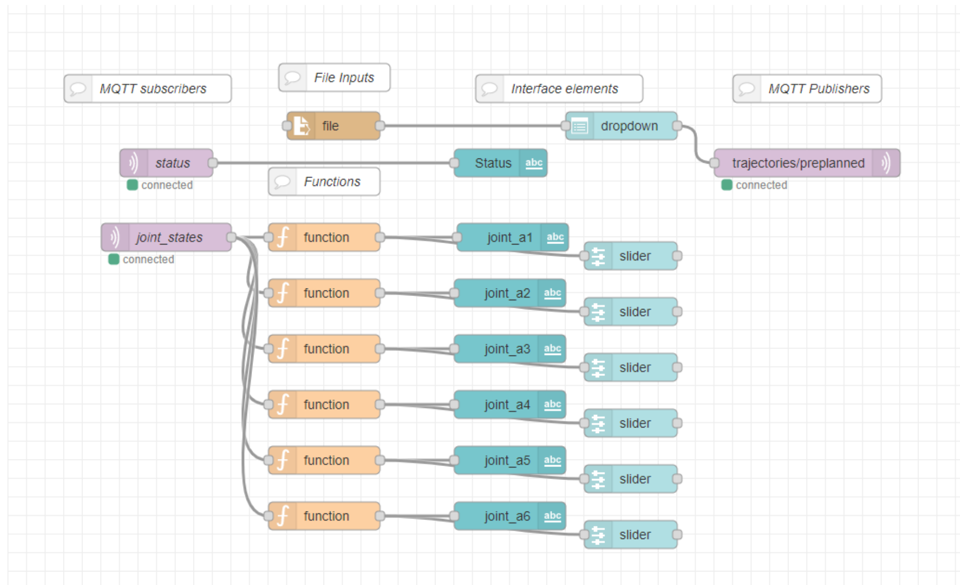


Figure 4 – Demonstration of the block-based programming of Node-RED

## 2.3 ROS framework

There is great correlation between MQTT and ROS, a framework that also uses a publisher-subscriber model to publish its messages. ROS is an open-source operating system intended for robotic systems [9]. Some of the advantages ROS provides are fast robot development and algorithm re-utilization. It

also provides a viable solution for Distributed Control Systems, allowing each robot to work modularly. It can work with very distinct programming languages. The common usage of robotic manipulators in aeronautical industry justifies its usage in this work.
In this work, we use the ROS Melodic Morenia distribution, compatible with Linux Ubuntu 18.04. We use packages developed in both Python and C++ programming languages, as well as XML for more native ROS applications.

#### *2.3.1 ROS-Industrial*

ROS-Industrial is a project that aims to create and advance practical applications of ROS in the industrial environment. It works as a consortium between researchers and industry members that develop many applications for the open-source project [10]. The repository has packages for many industrial robots, including experimental packages for KUKA manipulators, which will be used in this work.

### 2.4 AWS

AWS is an important tool for IoT technologies. It provides many services in cloud computing [11]. In this work, we are using it to provide an virtual machine. The machine will contain the bridge between the ROS core and the MQTT broker. To do so, We use the EC2 virtual machine.

### 2.5 Manipulator Robot

Manipulator robots are very common in industry usage, being vastly used for tool manipulation and production line operations. For the experiment, we use the manipulator robot KUKA KR16. In the experiment, we use the robot to simulate the usage of trajectory based tools. The intended use of the experiment is to simulate the action of milling tools involved in the manufacturing of aircraft body panels. However, in this configuration, our experiment can also represent a proper manipulator robot in a production line.

#### *2.5.1 Robot Sensor Interface*

There are three main interface that allow communication between a KUKA robot and a computer [12]. In this work, we utilise Robot Sensor Interface (RSI). Being an interfaces intended for sensor-assisted motion, it is very in line with the aims of this work, in which we will gather many of the sensor data to form a cloud database for our experiment.

## 3. Architecture

Figure 5 shows the proposed architecture for our experiment. On a local network, we have the HMI (human-machine interface), through which the operator can monitor and control the manipulator robot. In the local network we also have a MQTT Broker, which deals with the communication through MQTT protocol. This broker is connected to another MQTT broker on a remote computer in the cloud. This broker connects by a bridge with the ROS core in the EC2 computer on the cloud. The cloud computer connects to a ROS core in the robot computer by a bridge. The robot computer controls the KUKA KR16 robot performing the experiment.
The robot provides information regarding its operations and its sensors data to the computer, that them returns it to the cloud. As metrics send to the database, we intend to use time cycles, in relation to idle time and operation time. We also use velocity, vibration of the tool and stress on each axis. The KUKA KR16 robot used in this experiment has sensors for all these metrics.
One of the goals of this architecture is to facilitate the integration of different devices. Outside from the MQTT-ROS bridge, no device has to simultaneously deal with both MQTT and ROS communication. This would allow, for example, that all of the operations on the administrator network can be done in a microcontroller board or other low-resource device.

## 4. Implementation of the architecture

### 4.1 Interface

We use the node-red application to create a simple interface to be accessed in the administrator network, that can be accessed even through the cell phone. In the current implementation, this

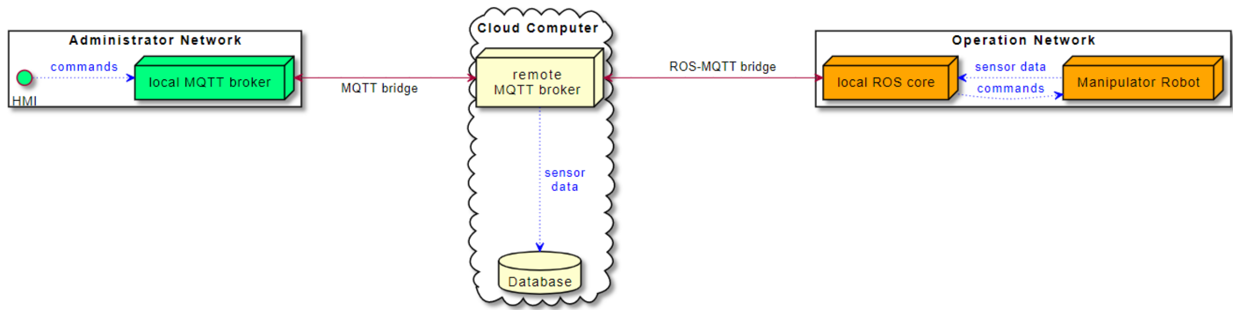


Figure 5 – Architecture of the experiment

interface is not fully independent. Due to the nature of the activities of what we want to do, the aim of this interface is mostly to monitor the current state of the manipulator robot and activate previously planned trajectories. The proper control of the trajectories is done with the use of a personal computer in the same network where the broker resides. An example of the interface can be seen in Figure 6.

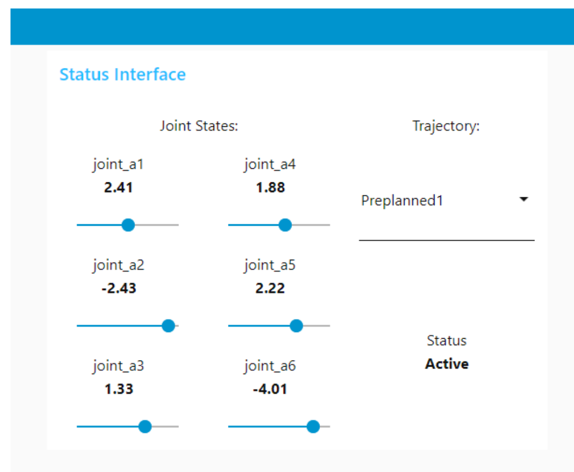


Figure 6 – Interface available to user

## 4.2 Bridge between MQTT Brokers

The mosquitto server can easily handle the bridge between two MQTT brokers, using only a configuration file. One of the brokers must be access to the IP of the second broker, the port of the communication must be defined, as well as the topics that will be part of the bridge. The code can be seen in Figure 7.

```
1 connection bridge-mqtt
2 address [ip]:1883
3
4 topic # out 0
5 topic # in 0
6
```

Figure 7 – Code of the configuration file of the bridge

As can be seen in the figure, the code defines the name of the connection (bridge-mqtt), the IP of the broker, the port in which the connection will be made (1883 is the standard port for unencrypted MQTT communication), and the definition of the topics which will be send and received. The usage

of "#" signifies that all topics in that broker will be transmitted. The number, in this case O, refers to the Quality of Service of the transmission.

## 4.3 ROS - MQTT Bridge

To make the connection between the MQTT Broker and the ROS core we used a bridge based on the open-source repository in [13]. The bridge is created by a ROS configuration file. The main difficult is the compatibility of the messages between MQTT and ROS. The application takes the messages from ROS, in its XML-based protocol, and and serializes it by json, making them MQTT compatible. In the same way, MQTT json messages are deserialized and made compatible with ROS messaging. The process is illustrated in Figure 8.

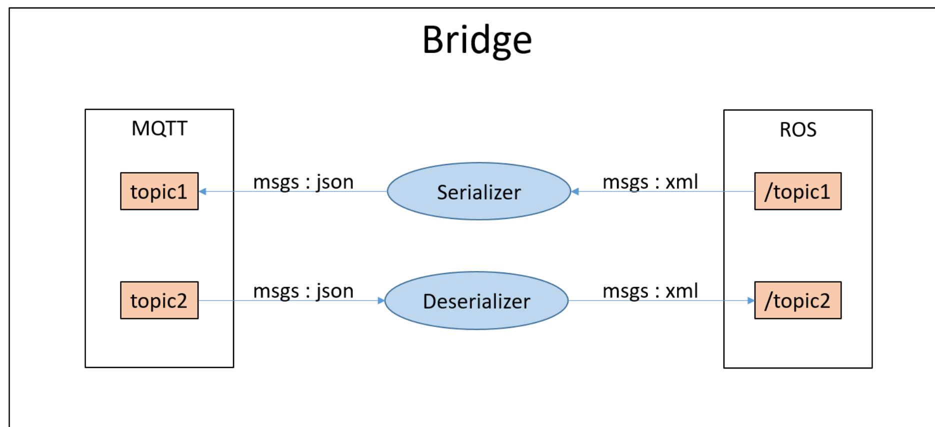


Figure 8 – Conversion of messages in the bridge

## 4.4 EC2 machine on AWS

We use EC2 machines to host the connections remotely. We are currently using a t2.micro machine with Linux Ubuntu 18.04 OS. Currently we are using the EC2 machine to handle communication with both the administrator and operator network. The servers in both networks connect to the EC2 machine by the bridges. In this implementation, the cloud machine is hosting a MQTT broker, but it is also able to host a ROS Core. We also use the machine to store the data from the robot operation.

## 4.5 Kuka_experimental

From the ROS-Industrial repository, we use the kuka_experimental package, which has experimental packages for many of the industrial robots of KUKA. The base application, that can perform the simulation of the robot in the native ROS visualizer, rviz, has the nodes and topics as presented in Figure 9.

Figure 9 – Base nodes and topics in the kuka_experimental package

### *4.5.1 URDF and XACRO*

For the proper utilization of the kuka_experimental package, it is required to have a model of the robot to use in its simulation. The model is made in the Unified Robot Description Format (URDF). URDF is a code-independent description of the geometry of the robot. Due to its large size, the URDF model is simplified by a XACRO (XML macro) file, becoming more compact. This allows for a visualization of the robot model using rviz, as can be seen in Figure 10.

Figure 10 – Model of the manipulator robot KUKA KR16 on rviz

## 4.6 Control of the Robot

We aim to use the robot to simulate the milling of aircraft body panels. To do so, the administrator network determines trajectories for the robot end-effector to perform. This set trajectories are transmitted then to the operation network. In this section, we describe the node organization of the ROS framework in order to perform this control, and give an slight oversight of how the control works.

### *4.6.1 ROS nodes*

The active nodes and topics in the ROS core can be seen in Figure 11. The commands arrive via the MQTT-ROS bridge to the node */mqtt_bridge*. The node publishes the commands in the topic */traj_set*, to which the */joint_client_py* node is subscribed. This node is responsible for performing the control of the robot. It interacts with the package */position_trajectory_controller*, publishing the desired joint states in the */follow_ joint_trajectory_goal* topic and awaiting for the confirmation of the movement by subscribing to the messages in the */follow_joint_trajectory_feedback* topic. This conversation is with the node */kuka_hardware_interface* which handles the interaction with the RSI that controls the robot. This node publishes the desired joint states, which will move the robot. The node /joint_state_publisher reads the desired states and publishes them to the visualizer, as seen in the base application ilustrated in Figure 9. The node /file_output_py is also reading the joint states, alongside with other data of the robot and publishing them back to the MQTT bridge that will publish them in the cloud network for storage.

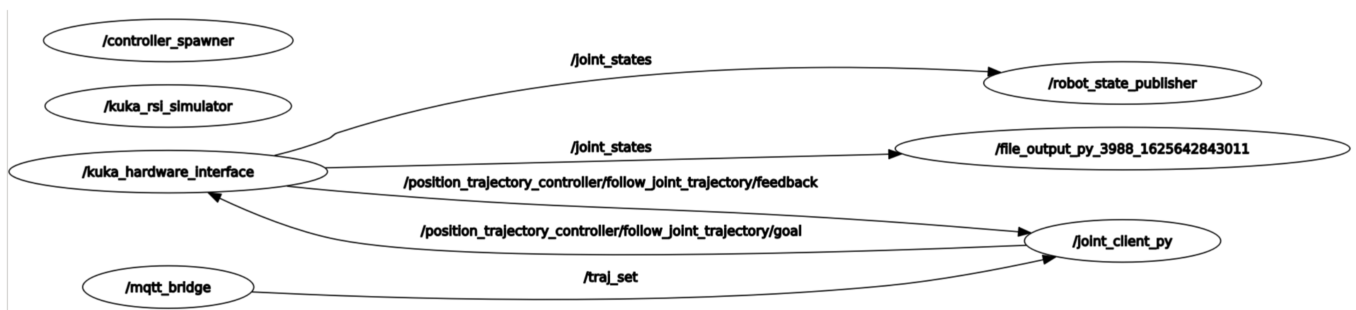


Figure 11 – Graph describing active ROS nodes and topics

### *4.6.2 Trajectory control*

The way that the node */kuka_hardware_interface* controls the robot is by defining its desired joint states. However, for our desired purposes of simulating the milling process of aircraft body panels, we require the ability to move the robot in linear trajectories, emulating the movement of the milling tool. To do this, we utilize the robotics toolbox from [15]. This toolbox allow us to create a model of the direct kinematics of the robot through the URDF file. With the direct kinematics, we can know which will be the position and orientation of the robot end-effector. Then, the toolbox allow us to calculate the inverse kinematics of the robot. With the inverse kinematics, we can determine the desired joint states in relation to the desired cartesian position that we want to achieve. The control then interpolates the initial and final states and creates a linear trajectory between both. All of this processes are made by the /joint_client_py node, but further detail of the control surpass the objectives of this paper.

## 5. Experiment

At this moment, we have all of the architecture described here already implemented, and working. In Figure 12, it is possible to see that the frequency of package delivery to the hardware interface is relatively stable. We will perform further tests regarding the stability of the network and the frequency in which packages are lost, and analyze if higher QoS are necessary for proper communication.
We are preparing to perform the physical experiment. We will use it to control, in the administrator network, the manipulator robot in the operation network. The robot will be doing trajectories compatibles with those of the milling tools used in the manufacturing process of aircraft body panels. We will evaluate the operation, and analyse the time cycles of the process. We will acquire data regarding velocity, vibration of the tool and stress on each axis of the robot during the trajectories, and store the data in the EC2 machine in the cloud.

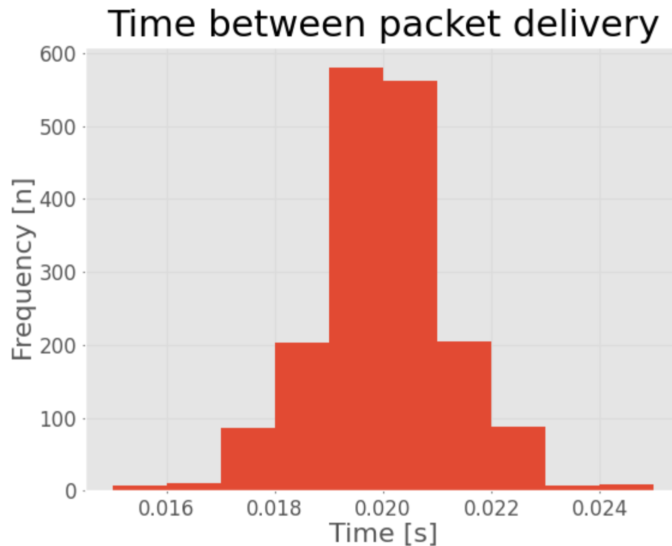


Figure 12 – Histogram of packet delivery between commands

## 6. Conclusion

In this work, we propose an experiment that utilizes many of current tools related to Industry 4.0 in an use case regarding the milling of aircraft body panels. We present an architecture for the experiment, describing each element necessary to the communication and control of the operation. We control a manipulator robot remotely, using a virtual machine in the cloud to manage the communication between the operation network and the administrator network. We use the ROS Framework to manage the communication robot-side, and the MQTT protocol on the other networks. The architecture proposed in this work can be a useful application for the manufacturing industry and is an important step in the direction of Industry 4.0 implementation, contributing to distributed and autonomous cyber-physical systems.

Further work is still necessary. We will perform the physical experiment and evaluate the results. In future implementations of the architecture, we plan to use more effective tools for data storage and the cloud, like AWS S3. We will also use the data collected to define autonomous responses for the system, aiming for preventive failure recognition, creating warnings regarding errors that can be identified by oversight operators.

## 7. Contact Author Email Address

To contact us, <mailto:leonardo.simiao.luna@usp.br>.

## 8. Copyright Statement

The authors confirm that they, and/or their company or organization, hold copyright on all of the original material included in this paper. The authors also confirm that they have obtained permission, from the copyright holder of any third party material included in this paper, to publish it as part of their paper. The authors confirm that they give permission, or have obtained permission from the copyright holder of this paper, for the publication and distribution of this paper as part of the ICAS proceedings or as individual off-prints from the proceedings.

## 9. Acknowledgements

Authors would like to thank AWS, CNPQ Grant 131354/2020-5, CAPES Grant 1825953, CEPOF, and FINEP, that partially funded this study.

## References

[1] Hermann M, Pentek T and Otto B. Design principles for industrie 4.0 scenarios. *49th Hawaii International Conference on System Sciences*, 2016.

[2] MQTT. MQTT: The Standard for IoT Messaging. <https://mqtt.org/>, accessed in February 25th, 2021.

[3] Thangavel D, et al. Performance evaluation of MQTT and CoAP via a common middleware. *2014 IEEE ninth international conference on intelligent sensors, sensor networks and information processing (ISSNIP)*, 2014.

[4] Light, R A. Mosquitto: server and client implementation of the MQTT protocol. *Journal of Open Source Software*, 2017.

[5] Singh, Meena, et al. Secure mqtt for internet of things (iot). *2015 fifth international conference on communication systems and network technologies.* IEEE, 2015.

[6] Lee, Shinho, et al. Correlation analysis of MQTT loss and delay according to QoS level. *The International Conference on Information Networking 2013 (ICOIN).* IEEE, 2013.

[7] Eclipse Foundation. Eclipse Mosquitto: An open source MQTT broker. <https://mosquitto.org/>, accessed in February 25th, 2021.

[8] Node-RED. Node-RED. <https://nodered.org/>, accessed in May 20th, 2021.

[9] ROS-Industrial. ROS-Industrial. <http://ros.org/>, accessed in February 25th, 2021.

[10] ROS. ROS. <https://rosindustrial.org/>, accessed in May 20th, 2021.

[11] Amazon. AWS. <https://aws.amazon.com/pt/>, accessed in February 25th, 2021.

[12] Arbo, M. H., et al. Comparison of KVP and RSI for Controlling KUKA Robots Over ROS. *IFAC-PapersOnLine 53.2* (2020).

[13] ROS. mqtt_bridge. <http://wiki.ros.org/mqtt_bridge>, accessed in February 25th, 2021.

[14] ROS-Industrial. kuka_experimental. <https://github.com/ros-industrial/kuka_experimental>, accessed in May 20th, 2021.

[15] Corke, P. and Haviland, J.. Not your grandmother's toolbox – the robotics toolbox reinvented for python. *IEEE International Conference on Robotics and Automation*, 2021.