

# A latency comparison of IoT protocols in MES

Erik Lindén

Jonas Wallgren  
Peter Jonsson

## 1.1 Upphovsrätt

Detta dokument hålls tillgängligt på Internet – eller dess framtida ersättare – under 25 år från publiceringsdatum under förutsättning att inga extraordinära omständigheter uppstår.

Tillgång till dokumentet innebär tillstånd för var och en att läsa, ladda ner, skriva ut enstaka kopior för enskilt bruk och att använda det oförändrat för ickekommersiell forskning och för undervisning. Överföring av upphovsrätten vid en senare tidpunkt kan inte upphäva detta tillstånd. All annan användning av dokumentet kräver upphovsmannens medgivande. För att garantera äktheten, säkerheten och tillgängligheten finns lösningar av teknisk och administrativ art.

Upphovsmannens ideella rätt innefattar rätt att bli nämnd som upphovsman i den omfattning som god sed kräver vid användning av dokumentet på ovan beskrivna sätt samt skydd mot att dokumentet ändras eller presenteras i sådan form eller i sådant sammanhang som är kränkande för upphovsmannens litterära eller konstnärliga anseende eller egenart.

För ytterligare information om Linköping University Electronic Press se förlagets hemsida <http://www.ep.liu.se/>.

## 1.2 Copyright

The publishers will keep this document online on the Internet – or its possible replacement – for a period of 25 years starting from the date of publication barring exceptional circumstances.

The online availability of the document implies permanent permission for anyone to read, to download, or to print out single copies for his/hers own use and to use it unchanged for non-commercial research and educational purpose. Subsequent transfers of copyright cannot revoke this permission. All other uses of the document are conditional upon the consent of the copyright owner. The publisher has taken technical and administrative measures to assure authenticity, security and accessibility.

According to intellectual property law the author has the right to be mentioned when his/her work is accessed as described above and to be protected against infringement.

For additional information about the Linköping University Electronic Press and its procedures for publication and for assurance of document integrity, please refer to its www home page: <http://www.ep.liu.se/>.

# Abstract

Many industries are now moving several of their processes into the cloud computing sphere. One important process is to collect machine data in an effective way. Moving signal collection processes to the cloud instead of on premise raises many questions about performance, scalability, security and cost.

This thesis focuses on some of the market leading and cutting edge protocols appropriate for industrial production data collection. It investigates and compares the pros and cons of the protocols with respect to the demands of industrial systems. The thesis also presents examples of how the protocols can be used to collect data all the way to a higher-level system such as ERP or MES.

The protocols focused on are MQTT and AMQP (in OPC-UA). The possibilities of OPC-UA in cloud computing is of extra interest to investigate in this thesis due to its increasing usage and development

# Table of contents

Chapter 1: Introduction.....	9
1.1 Introduction.....	9
1.2 Purpose.....	10
1.3 Problem Statement .....	10
1.4 Limitations .....	11
1.5 Report structure .....	11
Chapter 2: Background.....	12
2.1 PLC.....	12
2.2 SCADA systems.....	12
2.2.1 Field devices and signals.....	13
2.2.2 PLCs and RTUs .....	13
2.2.3 Supervisory servers .....	13
2.2.4 Operator clients (HMI).....	13
2.2.5 Communication network.....	14
2.3 DCS.....	14
2.4 MES.....	14
2.5 ERP .....	14
2.6 ICS.....	14
2.7 Cloud.....	15
2.8 Industry 4.0.....	15
2.9 Internet-of-Things .....	17
2.10 Summary.....	17
Chapter 3: Theory.....	18
3.1 Publish/Subscribe architecture .....	18
3.2 MQTT.....	18
3.2.1 Overview.....	19
3.2.2 Communication .....	19
3.2.3 Quality of service.....	22
3.2.4 Last will .....	22
3.2.5 Security.....	22
3.2.6 Wildcards.....	23
3.2.7 Summary.....	24
3.3 AMQP .....	24

3.3.1	Overview.....	24
3.3.2	Communication .....	25
3.3.3	Quality of service.....	26
3.3.4	Last will .....	26
3.3.5	Security.....	26
3.3.6	Wildcards.....	26
3.3.7	Summary.....	27
3.4	OPC-UA.....	27
3.4.1	Overview.....	27
3.4.2	OPC Classic.....	28
3.4.3	The address space .....	29
3.4.4	Services.....	29
Chapter 4:	Method.....	30
4.1	Introduction.....	30
4.2	Pre-Study .....	31
4.3	Implementation.....	31
4.3.1	MQTT .....	31
4.3.2	AMQP.....	32
4.3.3	Raspberry Pi.....	33
4.3.4	Windows 10 IoT Core .....	33
4.3.5	Software .....	33
4.4	Evaluation .....	34
Chapter 5:	Results .....	34
5.1	Introduction.....	34
5.2	MQTT .....	34
5.2.1	Major results .....	35
5.3	AMQP .....	35
5.3.1	Major results .....	35
5.4	Latency comparison .....	36
5.5	Features.....	36
Chapter 6:	Discussion .....	37
Chapter 7:	Conclusions.....	37
7.1	Future work .....	38
Chapter 8:	Bibliography.....	39

# Illustration Index

- Figure 1 The traditional communication pyramid – factory floor to enterprise..... 9
- Figure 2 Cloud communication within the classical communication pyramid..... 10
- Figure 3 Typical classical SCADA architecture ..... 13
- Figure 4 Industrial revolutions ..... 16
- Figure 6 Publish-subscribe pattern, as illustrated by commercial messaging ecosystem HiveMQ ..... 18
- Figure 7 MQTT Connect pattern..... 19
- Figure 8 MQTT Publish pattern ..... 20
- Figure 9 MQTT Subscribe pattern ..... 21
- Figure 10 Quality of service levels of MQTT..... 22
- Figure 11 MQTT Wildcard example setup..... 24
- Figure 12 OPC-UA in the classical communication pyramid ..... 28
- Figure 13 Simple OPC-UA address space..... 29
- Figure 14 Address space as seen in commercial OPC-UA client “UaExpert” from Unified Automatio.29
- Figure 15 MQTT test bench setup ..... 32
- Figure 16 AMQP/OPCUA test bench setup ..... 32
- Figure 17 MQTT Latency measurement results ..... 35
- Figure 18 AMQP Latency measurement results..... 36
- Figure 19 MQTT and AMQP Latency comparison ..... 36
  
- Table 1 MQTT Connect parameters ..... 20
- Table 2 MQTT Publish parameters..... 21
- Table 3 MQTT Subscribe parameters ..... 21
- Table 4 Quality of Service modes in MQTT ..... 22
- Table 5 AMQP Exchange types..... 27
- Table 6 OPC-UA standard services ..... 30
- Table 7 Test bench hardware comparison with PLC ..... 33
- Table 8 MQTT major results ..... 35
- Table 9 AMQP major results..... 35

## List of abbreviations and acronyms

<b>Abbreviation/ Acronym</b>	<b>Meaning</b>	<b>Explanation</b>	<b>Context</b>
AMQP	Advanced Message Queuing Protocol	Open standard application layer protocol for message-oriented middleware.	One of the protocols described and evaluated in the project, described in chapter 3
ARM	Advanced RISC Machine	A family of reduced instruction set computing (RISC) architectures for computer processors,	Architecture used in Raspberry Pi, which is the device that acted as a simulated PLC for this project
CPS	Cyber Physical System	The tight conjoining of and coordination between computational and physical resources	Described in chapter 2
DCS	Distributed Control System	Computerised control system	Described in chapter 2
ERP	Enterprise resource planning	Business management software	Described in chapter 2
HMI	Human Machine Interface	The space where interactions between humans and machines occur. Often a specific device or screen.	Described in chapter 2
ICS	Industrial Control System	Computer systems and networks used to control industrial plants and infrastructures	Described in chapter 2
IIoT	Industrial Internet of Things	The infrastructure of the industrial information society	Described in chapter 2
IoT	Internet of Things	The infrastructure of the information society	Described in chapter 2
M2M	Machine-to-Machine	Communication between devices using any communications channel	Described in chapter 2
MES	Manufacturing Execution System	Computerized system used in manufacturing	Described in chapter 2

MQTT	Message Queuing Telemetry Transport	Publish-subscribe-based messaging protocol	One of the protocols described and evaluated in the project, described in chapter 3
OPC	Open Platform Communications	A series of standards and specifications for industrial communication.	Described in chapter 3
OPC-UA	Open Platform Communications Unified Architecture	Industrial M2M communication protocol for industrial interoperability.	Described in chapter 3. AMQP is used and evaluated within OPC-UA
PLC	Programmable Logic Controller	Industrial digital computer	Described in chapter 2
QoS	Quality of Service	The overall performance of a telephony or computer network	Used in MQTT
RTU	Remote Terminal Unit	Microprocessor-controlled electronic device used in SCADA	Described in chapter 2
SCADA	Supervisory Control And Data Acquisition	A control system architecture	Described in chapter 2
SDK	Software Development Kit	A set of low-level functions to facilitate development of more complex software	Azure SDK was used for the test implementations of this project, described in chapter 5.
TCP	Transmission Control Protocol	Transport-layer protocol for wired and wireless communication	Used in MQTT and AMQP
UWP	Universal Windows Platform	Platform-homogeneous application architecture created by Microsoft and first introduced in Windows 10	The architecture of the publisher program. This was used in order to deploy the programs on Windows IoT Core

# Chapter 1: Introduction

## 1.1 Introduction

In the traditional communication pyramid point of view of industrial communication, large amounts of data need to be transferred between different layers.

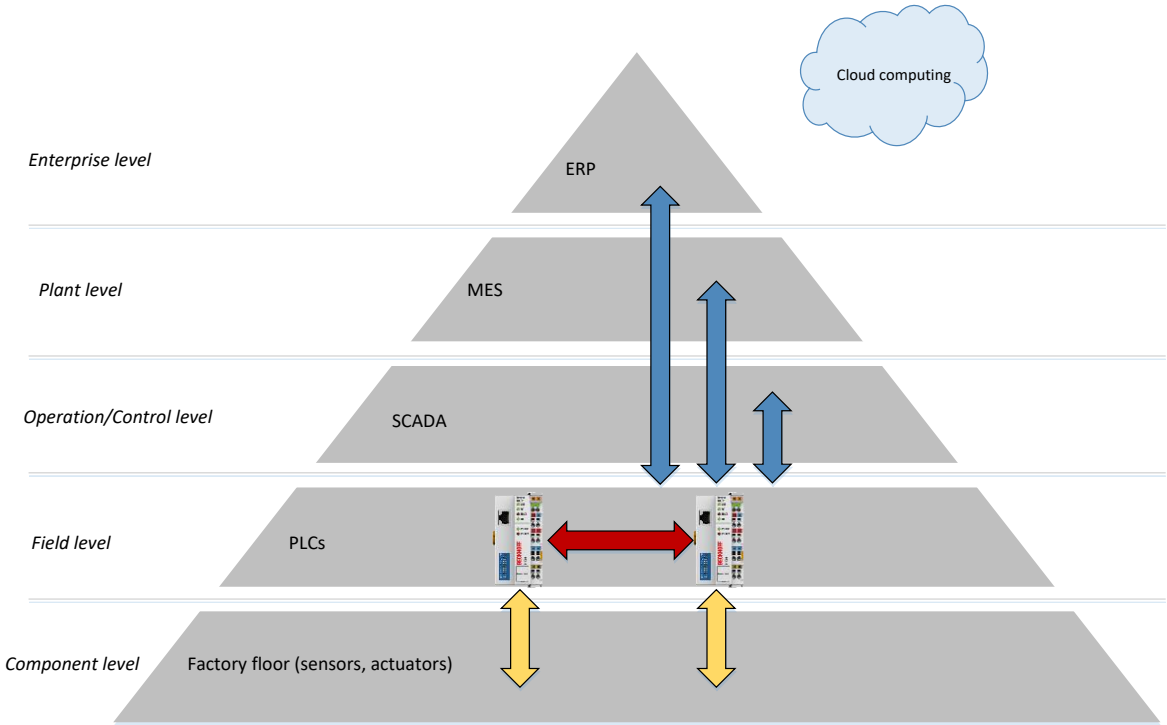


Figure 1 The traditional communication pyramid – factory floor to enterprise

In the classic scenario, a manufacturing plant has a SCADA/DCS system to collect data from PLCs which in turn are connected to sensors (yellow arrows in figure above). Machine-to-Machine (M2M) communication may also occur (red arrows). The transfer of data to higher-level implementations is often communicated directly from PLCs with vendor specific protocols (blue arrows). While PLCs and field level sensors are often included in the term SCADA, they are here depicted as a separate level of abstraction.

With Industry 4.0 and the rise of Industrial Internet of Things (IIoT) PLCs can directly be connected to the cloud, splitting the traditional communication pyramid:

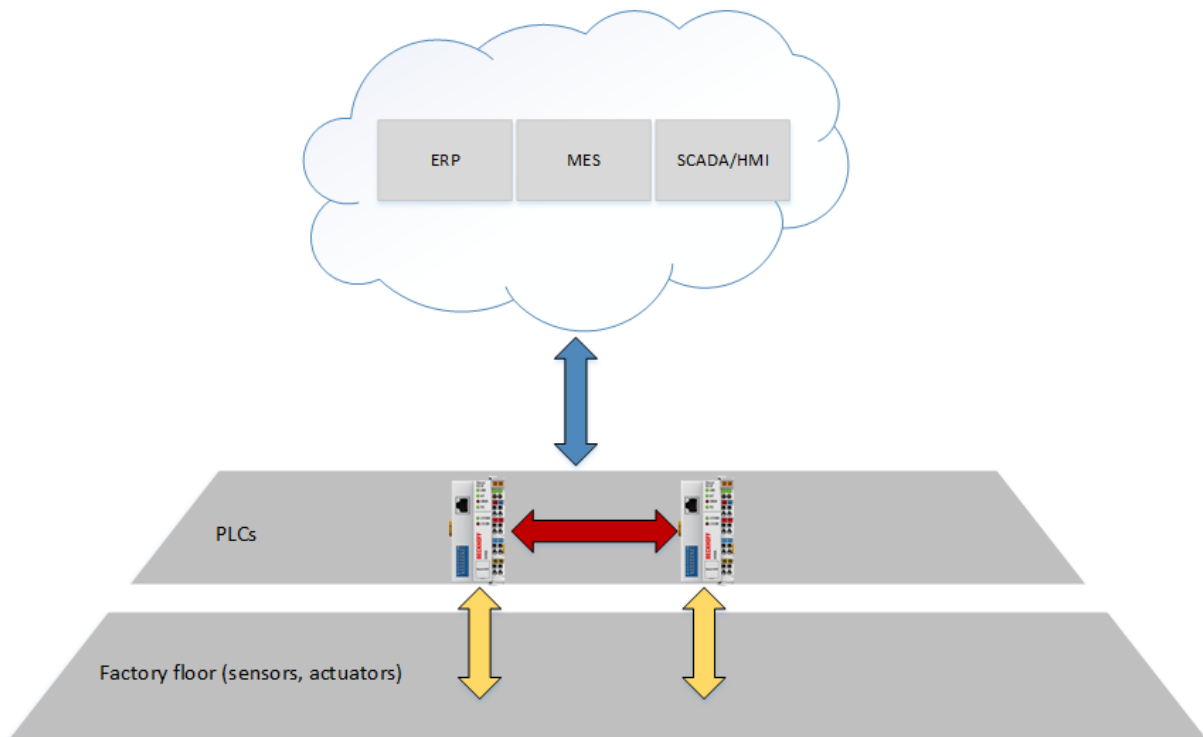


Figure 2 Cloud communication within the classical communication pyramid

## 1.2 Purpose

It is important to examine the business objectives that should be reached when creating or moving existent applications to the cloud. This includes establishing the benefits to be gained as a company. With cloud computing in general, there is an obvious interest in reducing in-house hardware, as well as the security and scalability aspects of using commercial cloud services instead of private servers.

In this project, the main purpose is for MES customers to move their data collection to the cloud. Enabling them to only possess “PLC-islands” in the factory, i.e. standalone devices that are only connected to machine sensors and an internet connection. This gives the customers the possibility to discard private servers and complex IT infrastructure.

The cloud computing also gives MES software vendors the possibility to scale and offer MES solutions as “Software as a Service” (SaaS). Which can greatly reduce costs and facilitate software deployment.

## 1.3 Problem Statement

In the eyes of a MES vendor the main question is: which protocols shall we use to transfer data from the PLC to the cloud (and in turn the higher-level software solution)? Many PLC vendors offer many different solutions – including vendor-specific protocols and solutions. This thesis investigates two of the major protocols that have been promoted by several different PLC vendors such as Siemens, Beckhoff and WAGO. The thesis tries to answer how MQTT and AMQP (in OPC-UA) can be used in a cloud-environment to transfer production data from the factory floor to top level systems. As well as what are their respective limitations and pros and cons.

One of the most important factor when sending input-data to a MES is the latency. This thesis focuses on the latency (and to lesser extent features) and compares the latency of the two different protocols when used in a MES setup.

The questions this thesis tries to answer are:

- What latency could a cloud-ready MES implementation expect from AMQP and MQTT?
- How much will the latency differ between MQTT and AMQP with respect to message reliability?
- How much will the latency differ between MQTT and AMQP with respect to message size?

The study of the features of the protocols, in combination with the measurement of the latencies will hypothetically give an understanding of why organizations such as OPC Foundation choose to implement AMQP in the OPC-UA stack rather than MQTT.

## 1.4 Limitations

The question of which protocol to use is specifically asked with respect to MES systems, i.e. the focus is not M2M communication or real-time applications, but rather slower-paced data streams.

Due to the fact that many PLC vendors are still in the process of developing cloud solutions (as of November 2016, most PLC vendors have not yet released cloud-ready devices or PLC code to be used to communicate with the cloud).

This fact limits the project in such that the implementations test of the different protocols cannot be tested on industry-ready PLCs. The implementations and tests will instead be evaluated on devices with similar hardware. Production data with realistic signals will be simulated with software.

Another limitation is that the main comparison is made by investigating MQTT and AMQP (in OPC-UA). These are only two protocols of many used in IoT-applications (such as DDS, CoAP and XMPP).

## 1.5 Report structure

The report is structured in the following manner:

Chapter 1: Introduces the project and establishes the problem to be solved.

Chapter 2: Introduces the systems and components interacting in a plant, as well as the recent trends in industrial systems.

Chapter 3: Describes details about the protocols tested in the thesis.

Chapter 4: Describes how the protocols can be used in order to collect industrial data to the cloud and presents two example tests.

Chapter 5: Describes the results of the tests.

Chapter 6: Discussion chapter.

Chapter 7: Discussion of cases and situations where one protocol should be used over the other.

# Chapter 2: Background

In order to get a clear view of bottom-to-top data collection, a description of the different components of the systems used in a factory are given. The description starts from the bottom layer by describing systems such as SCADA and DCS, and ends by describing ICS, which is a collective term for a general system. The last parts of the chapter describe how cloud computing, IoT and Industry 4.0 concepts interact with more classical systems.

## 2.1 PLC

PLCs are digital electronic devices with a programmable memory for storing instructions to implement logic, sequencing, timing, counting, and arithmetic functions. Since its inception in 1969, the PLC consists of four major components: processor unit, input modules, output modules and programming device.

The basic operation principle of a PLC is program scanning. A PLC processor executes a program in a cyclic manner. During scanning, the processor simultaneously updates the status of input and output instructions in both the program and the memory, as well as interacting with input and output modules, in turn affecting devices connected to the PLC.

The benefits of using PLCs are obvious. They are flexible and programmable, reliable and cost saving. PLCs provide ease of installation and implementation, as well as ease of maintenance and troubleshooting. [1]

## 2.2 SCADA systems

SCADA (Supervisory Control and Data Acquisition) systems incorporate Programmable Logic Controllers (PLCs), Human Machine Interface (HMI) workstations and network communication systems into a complete integrated system.

The typical SCADA system today consists of five major types of components:

- Field devices and signals
- Programmable logic controllers (PLC) and Remote Terminal Units (RTU)
- Supervisory servers
- Operator clients (HMI)
- Communication network

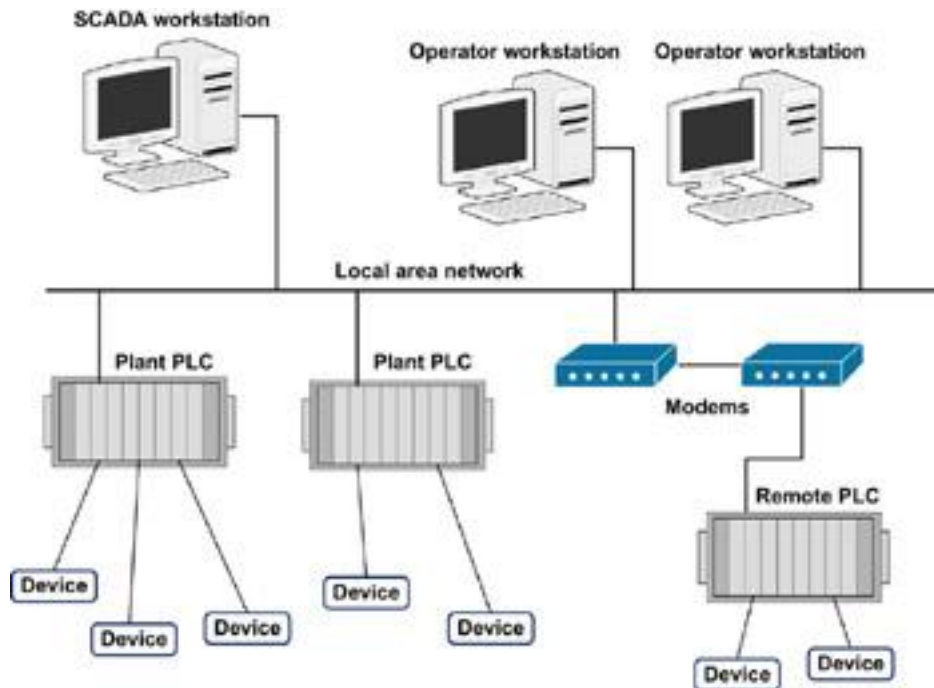


Figure 3 Typical classical SCADA architecture [1]

### 2.2.1 Field devices and signals

This is the lowest level in the SCADA system. The field devices are represented as signals into and out of the PLCs and RTUs. This includes controlling devices, such as pumps, valves, solenoids, etc. These field devices are the link between the SCADA technology and the process operations of any automated facility. [1]

### 2.2.2 PLCs and RTUs

Core components of the SCADA system, the PLCs and RTUs are connected to the field devices and signals. And themselves connected to the supervisory servers. The PLCs specifically also have a wide programming ability in languages such as IEC 61131-3 programming language. [1]

### 2.2.3 Supervisory servers

The supervisory servers are responsible for communicating with the RTUs and PLCs, and include the HMI software running on operator workstations. There can be many distinctions of how the supervisory server is set-up, depending on the size of the SCADA system. In a smaller SCADA system, the supervisory computer may be composed of a single PC, on which the HMI is installed or a component. In larger SCADA systems, the master server may include several HMIs hosted on client computers, multiple servers for data acquisition and distributed software applications. [1]

### 2.2.4 Operator clients (HMI)

The HMI (Human to Machine Interface) is the graphical user interface with process graphic displays, trends and associated reports. [1]

### 2.2.5 Communication network

The communication network is the hardware and software that interconnect all of the components of the system. Typical networks today include Ethernet with Transmission Control Protocol/Internet Protocol (TCP/IP) and proprietary communication topology. [1]

## 2.3 DCS

The border between a SCADA and DCS (Distributed Control System) is very vague and often the terms are used interchangeably. It is not uncommon that tasks can be performed both by a SCADA and a DCS system, however in industrial reality few usages have been designed with this in mind. As an example, a DCS system has a process oriented view, while a typical SCADA system is more data acquisition oriented. Generally, SCADA systems are expected to operate reliably over unreliable links (but always keep a backlog of acquired data), while DCS systems have direct access to the source of data and therefore the latest values. Additionally, SCADA systems are mostly event-driven while DCS systems generally run sequentially. [2]

## 2.4 MES

A MES (Manufacturing Execution System) can be viewed as a higher-level system than a SCADA system. A key component in a MES system is the management of resources. MES provides the ability to control machines, labor skills, materials, and documents among other resources necessary for an operation to be performed. History of resources, current setup, availability, and other critical information is simultaneously available to the technician on the shop floor and the manager.

Another key component of a MES is the scheduling of work orders. Sequencing of work based on priority, attributes, and resource requirements seeks to minimize setup time and maximize flow through the production system. An accurate calculation of time spent is compiled from each independent operation even with the added complexity of overlapping or parallel operations. The scheduling feature of MES also provides for level loading of labor and equipment. Data collected from a SCADA/DCS system are generally used in a MES. [1]

## 2.5 ERP

The study of ERP (Enterprise Resource Planning) systems is out of scope for this thesis. However, an ERP system is often the endpoint of the data collected by a SCADA or DCS system. Often used in parallel with a MES. [3]

## 2.6 ICS

ICS (Industrial control system) is a general term that encompasses several types of control systems, including supervisory control and data acquisition (SCADA) systems, distributed control systems (DCS), and other control system configurations such as Programmable Logic Controllers (PLC) often found in the industrial sectors and critical infrastructures. An ICS consists of combinations of control components (e.g., electrical, mechanical, hydraulic, pneumatic) that act together to achieve an industrial objective (e.g., manufacturing, transportation of matter or energy). The part of the system primarily concerned with producing the output is referred to as the process. The control part of the system includes the specification of the desired output or performance. ICS controlled industrial processes are typically used in electrical, water and wastewater, oil and natural gas, chemical,

transportation, pharmaceutical, pulp and paper, food and beverage, and discrete manufacturing (e.g., automotive, aerospace, and durable goods) industries. [4]

## 2.7 Cloud

The so-called public cloud service providers, e.g. Microsoft Azure™ or Amazon Web Services™ (AWS), provide users with a range of services from their own data centers. This includes virtual machines, where the actual user has general control of the operating system and the applications installed on it, and stretches to abstracted communication and data services, which can be integrated by the user in an application. An example of the latter is the Azure IoT Hub: which is a cloud component that can act as a standalone message broker. This means that the broker will run in the cloud standalone and do not need a separate own virtual machine. Note that the Virtual Machine and Azure IoT Hub used in this thesis is an extremely small subset of the whole Azure ecosystem.

## 2.8 Industry 4.0

The German term Industry 4.0 describes the increased integration of information and communication technologies into production. In spring 2014, VDMA, Bitkom and ZVEI, three leading German associations of mechanical engineering, information, communication and electrical industry, released a definition for Industry 4.0. According to them, Industry 4.0 aims for optimization of value chains by implementing an autonomously controlled and dynamic production. Enablers are the availability of real time information and networked systems. Instruments to reach this increased automation are Cyber Physical Systems (CPS). Equipped with microcontroller, actuators, sensors and a communication interface, CPS can work autonomously and interact with their production environment. As a result, a factory becomes 'smart'. [5]

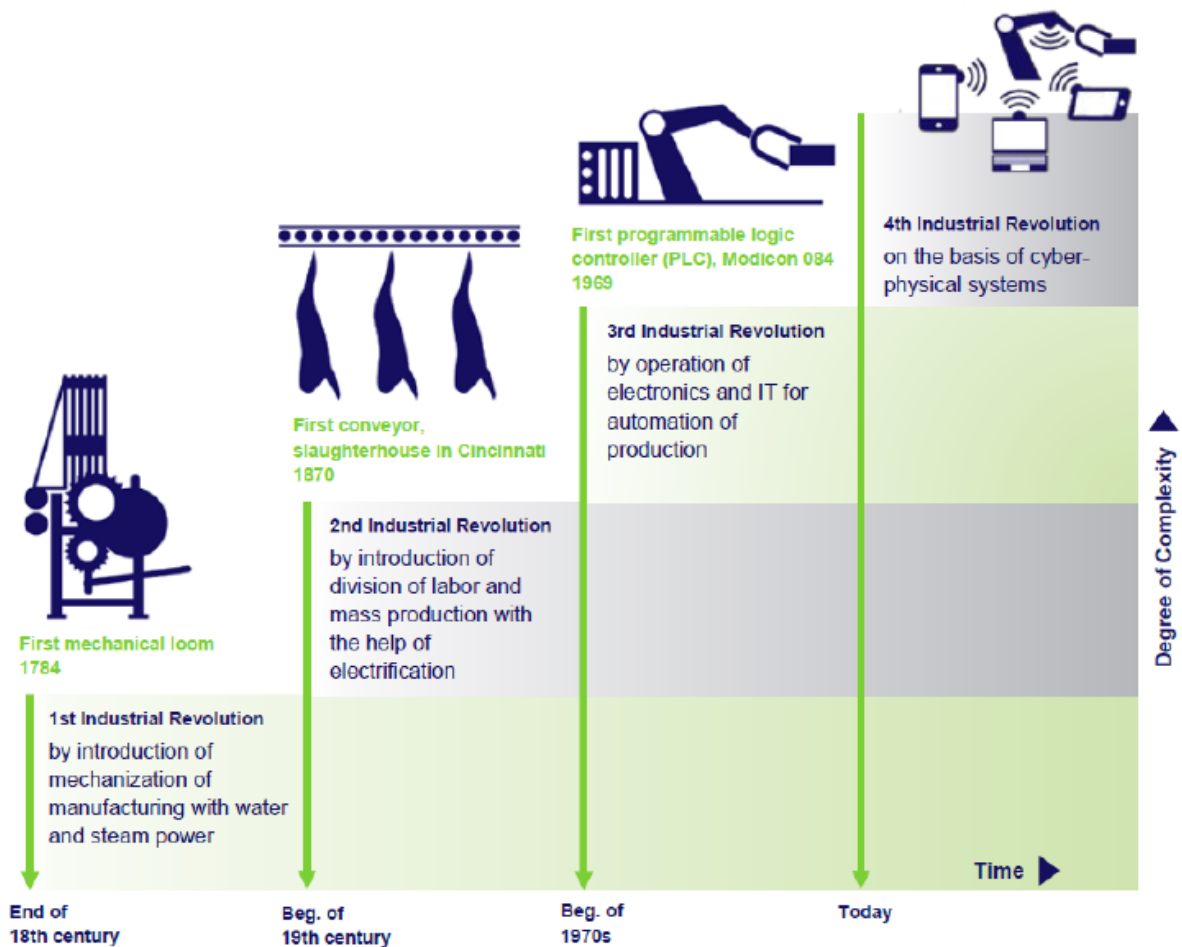


Figure 4 Industrial revolutions [7]

Industry 4.0 demands flexibility, adaptability, transparency and much more requirements which have to be fulfilled by Industry 4.0 components or systems. The Implementation strategy of the German Initiative Industry 4.0 names several existing and usable approaches or technologies for individual aspects which are candidates for Industry 4.0 standards. Due to the complexity of Industry 4.0, there is no single standard, but the need to integrate and combine different standards from different domains which cover different aspects. OPC-UA was named with good prospects for the communication aspect. The variability and flexibility of OPC-UA range from simple process data acquisition to complex monitoring, control, and analysis. It covers security aspects and provides the ability to represent and cover semantics by its information model. Additionally, OPC-UA is explicitly open for the integration and combination with other standards based on so called companion specifications. [6]

The data collected by these networks will be represented virtually and the processes will be controlled remotely. CPS are working as a system by definition and forming a part of what is often referred to as the Internet of Things (IoT). The IoT can be understood as a data and information cloud that is conceptually quite similar to that of CPS in that it consists of embedded systems

communicating through a network. However, the IoT has a broader scope since its components can be functioning independently and not only include CPS. [7]

## 2.9 Internet-of-Things

The Internet of Things (IoT) is a new technology paradigm envisioned as a global network of machines and devices capable of interacting with each other. The true value of the IoT for enterprises can be fully realized when connected devices are able to communicate with each other and integrate with higher level systems such as ERP, MES, customer support systems, business intelligence applications, and other business analytics systems. The IoT is recognized as one of the most important areas of future technology and is gaining vast attention from a wide range of industries.

Some forecast suggests that the IoT concept will reach 26 billion units by 2020, up from 0.9 billion in 2009, and will impact the information available to supply chain partners and how the supply chain operates. [8]

From production line and warehousing to retail delivery and store shelving, the IoT is transforming business processes by providing more accurate and real-time visibility into the flow of materials and products. Firms will invest in the IoT to redesign factory workflows, improve tracking of materials, and optimize distribution costs. Several companies and organization have already implemented large scale IoT projects. For instance, UPS is already using IoT-enabled fleet tracking technologies to cut costs and improve supply efficiency. [9]

Internet of Things generally refers to scenarios where network connectivity and computing capability extends to objects, sensors and everyday items not normally considered computers, allowing these devices to generate, exchange and consume data with minimal human intervention.

Considering that the paradigm IoT encompasses a broad range of applications. A paradigm closer to the industrial and manufacturing world is IIoT (Industrial Internet-of-Things).

The IIoT is an innovative technology, directly interconnecting a set of sensors and devices (such as PLC's) to collect, record, transmit, and share data for possible analysis. The term is simply used when to describe IoT applications related to the production of goods and services, including in manufacturing and utilities. [10]

## 2.10 Summary

In this chapter, we have seen the relationships between the different systems in a factory. We have seen how they interact with each other and what functionality each system is responsible for.

# Chapter 3: Theory

In the previous chapter, basic communication patterns and how interaction from a factory-floor to a top-level system are performed were presented. This chapter will introduce the concept of publish/subscribe architecture and how it can be used in a SCADA/DCS system to collect data. MQTT and AMQP are described in details in this chapter. The chapter ends with a description of the OPC-UA technology.

## 3.1 Publish/Subscribe architecture

The publish/subscribe pattern is based on a client/server architecture where a central server, also known as a broker, receives messages from the clients. The clients are essentially all the nodes involved in the communication scheme. Each message is sent on a specific topic and a message can either be a topic publication or subscription. The publish/subscribe pattern enables many-to-many communication and simply decouples producers and consumers. This flexibility and simplicity facilitate the connection of devices to middleware and other applications. [11]

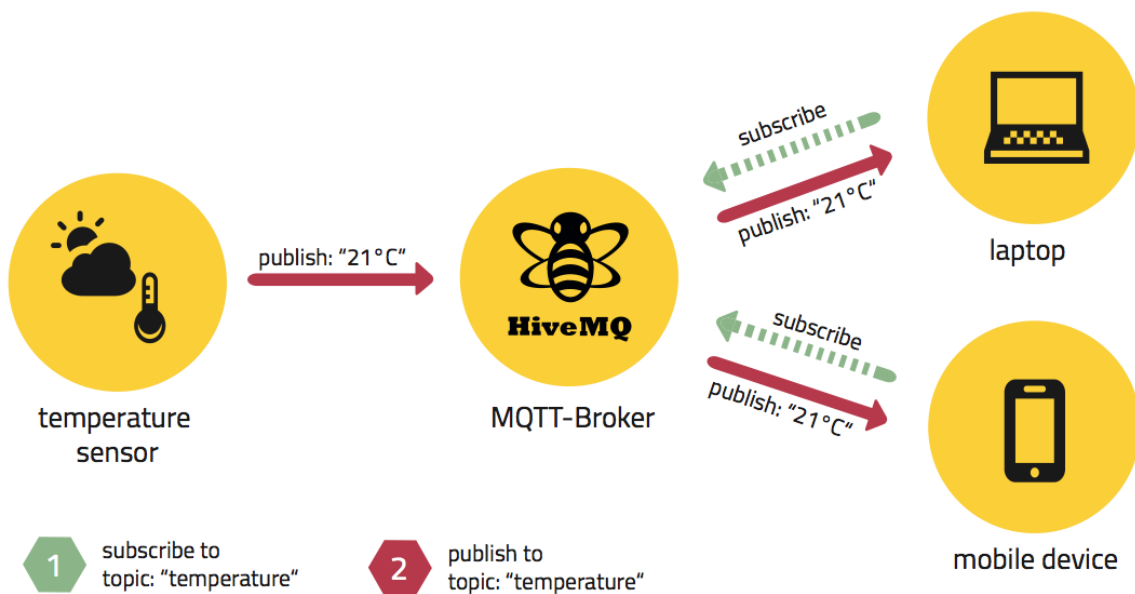


Figure 5 Publish-subscribe pattern, as illustrated by commercial messaging ecosystem HiveMQ. [15]

## 3.2 MQTT

### 3.2.1 Overview

MQTT (Message Queuing Telemetry Transport) is a publish/subscribe protocol running on top of TCP and was originally developed by IBM. In 2013, it was turned over to the OASIS (Organization for the Advancement of Structured Information Standards). The current OASIS standard version of MQTT is 3.1.1 [12].

MQTT was designed to be a lightweight, open, simple, and easy to implement protocol which would make it ideal for use in the context of Machine to Machine (M2M) communication and IoT where bandwidth and battery power are important factors.

MQTT is so lightweight that it can be supported by some of the smallest measuring and monitoring devices, and it can transmit data over far reaching, sometimes intermittent networks. It is also architected to overcome the challenges of connecting the expanding physical world (as covered in the previous chapter) of sensors, actuators, phones, and tablets with established software processing technologies. MQTT transports opaque and do not define a mechanism to express the payload encoding. MQTT can also use Websockets for its communication. [13]

### 3.2.2 Communication

The methods defined by MQTT are (note that in the illustration below the broker is in the PLC and cloud environment. MQTT do not define device or implementation used. Note that the index within the parenthesis indicates the event's order with respect to time): [14]

#### 3.2.2.1 Connect

A CONNECT packet which sends identifying information about the client to the broker will be sent directly after a network connection has been established between a client and a broker. The broker response with a CONACK and a status code. Once the connection is established, the broker will keep it open as long as the client does not send a disconnect command or it loses the connection. [15]

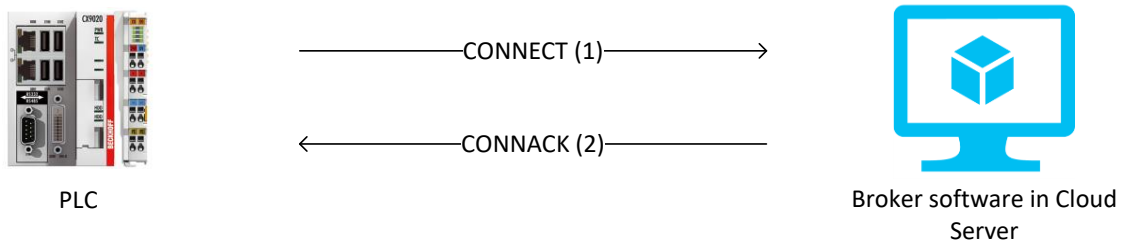


Figure 6 MQTT Connect pattern

The Connect packet shall be supplied with a set of parameters, the parameters that can be supplied are:

Name	Description
clientId	A unique ID each MQTT client connecting to a MQTT broker. The broker uses this ID in order

	to identify the client and the current state of the client
cleanSession (optional)	The clean session flag indicates the broker, whether the client wants to establish a persistent session or not. A persistent session (CleanSession = false) means, that the broker will store all subscriptions for the client and also all missed messages, when subscribing with Quality of Service (QoS) 1 or 2. If clean session is set to true, the broker won't store anything for the client and will also purge all information from a previous persistent session.
Username (optional)	Username in plaintext
Password (optional)	Password in plaintext
lastWillTopic (optional)	Last Will topic, see section below for details.
lastWillQos (optional)	Last Will QoS, see section below for details.
lastWillMessage (optional)	Last Will message, see section below for details.
lastWillRetain (optional)	Last Will reation , see section below for details.
keepAlive	The keep alive is a time interval, the clients commits to by sending regular PING Request messages to the broker. The broker response with PING Response and this mechanism will allow both sides to determine if the other one is still alive and reachable.

Table 1 MQTT Connect parameters

### 3.2.2.2 Disconnect

Waits for the client to finish its work and for the TCP session to terminate.

### 3.2.2.3 Publish

Sends a PUBLISH message from a client to a broker or from a broker to a client to transport a message.

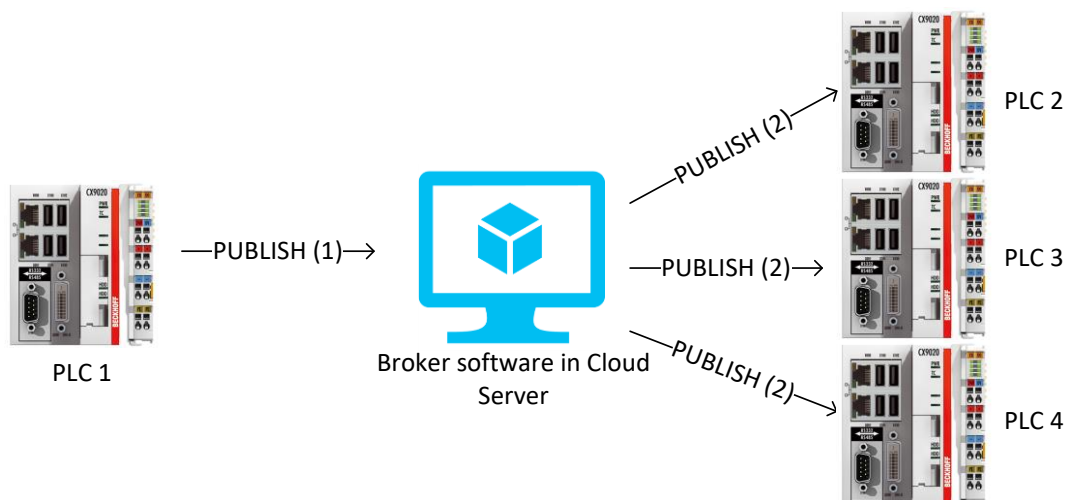


Figure 7 MQTT Publish pattern

Name	Description
packetId	The packet identifier is a unique identifier between client and broker to identify a message.
topicName	A simple string, which is hierarchically structured with forward slashes as delimiters. Example: "sweden/linkoping/temperature".
qos	Quality of Service Level for this message, see section below for details.
retainFlag	This flag determines if the message will be saved by the broker for the specified topic as last known good value. New clients that subscribe to that topic will receive the last retained message on that topic instantly after subscribing. More on retained messages and best practices in one of the next posts.
payload	Message content.
dupFlag	The duplicate flag indicates, that this message is a duplicate and is resent, because the other end did not acknowledge the original message. This is only relevant for QoS greater than 0.

Table 2 MQTT Publish parameters

[16]

### 3.2.2.4 Subscribe

Sends a SUBSCRIBE message from a client to a broker in order to create subscriptions for one or more topics.

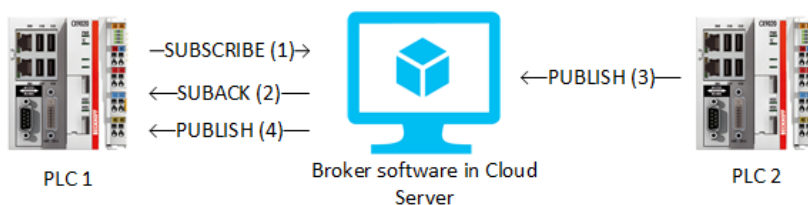


Figure 8 MQTT Subscribe pattern

[16]

The subscribe method is supplied by a packet id and a list of subscriptions, where N in the table below is an integer:

Name	Description
packetId	Packet identifier
qos N	Quality of Service for the subscription
topic N	Topic for the subscription.

Table 3 MQTT Subscribe parameters

### 3.2.2.5 Unsubscribe

Sends an UNSUBSCRIBE message from the client to the broker to unsubscribe from one or more topics

### 3.2.3 Quality of service

For reliability, MQTT offers three types of modes which are called Quality of Service (QoS). The QoS are specified when publishing a message. [17].

Quality of Service mode	Result
0 (At most once)	Sends a packet only one time without requiring confirmation messages or ACK. Even though QoS 0 is used which does not require MQTT ACK responses, TCP still provide TCP ACK for every package sent. This is also sometimes referred to “Fire and forget”
1 (At least once)	Ensures that the message is delivered at least once by requiring an ACK
2 (Exactly once)	Guarantees that the message is delivered exactly once

Table 4 Quality of Service modes in MQTT

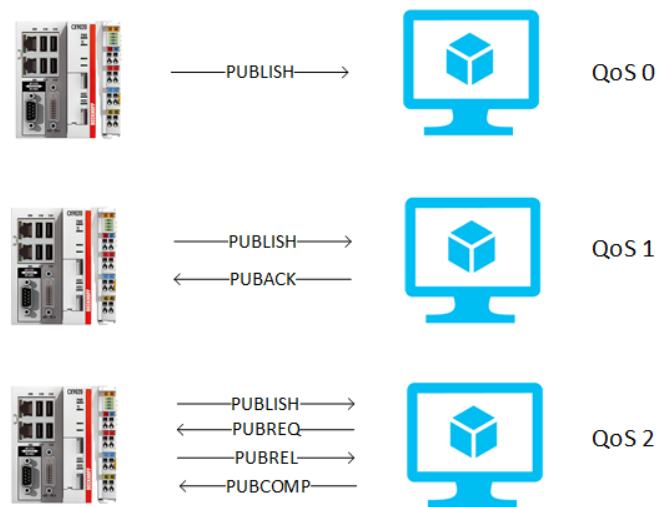


Figure 9 Quality of service levels of MQTT.

### 3.2.4 Last will

The client can also specify a so called “Last Will” message upon connection. If a client has not published any messages during 1.5 times the Keep Alive time the broker will disconnect the client and publish the last will on the topic specified upon connection from the client. [14]

### 3.2.5 Security

#### 3.2.5.1 Authentication of clients by the server

A CONNECT Packet contains Username and Password fields. Implementations can use these fields to supply credentials to the server. Implementations may as well provide their own authentication mechanism, use an external authentication system such as LDAP (Lightweight Directory Access Protocol) or OAuth (an open standard for authorization) tokens, or leverage operating system authentication mechanisms. [14]

### 3.2.5.2 Authentication of the server by the client

The MQTT protocol itself is not trust symmetrical: it provides no mechanism for a client to authenticate the server. [14]

### 3.2.6 Wildcards

There are two wildcard characters which can be used in MQTT, + and #.

The character '+' matches any topic on a single hierarchical level, and the character '#' matches any number of levels.

For instance, a global temperature database might subscribe to sensors/temperature/# and it would receive temperature readings from every sensor in the world.

However, if for instance the Swedish government wanted to use the data for their own weather service, they could just subscribe to sensors/temperature/se/#, thereby limiting the sensor readings to those within the Sweden. Naturally, the usage of wildcard patterns is completely implementation specific. [14]

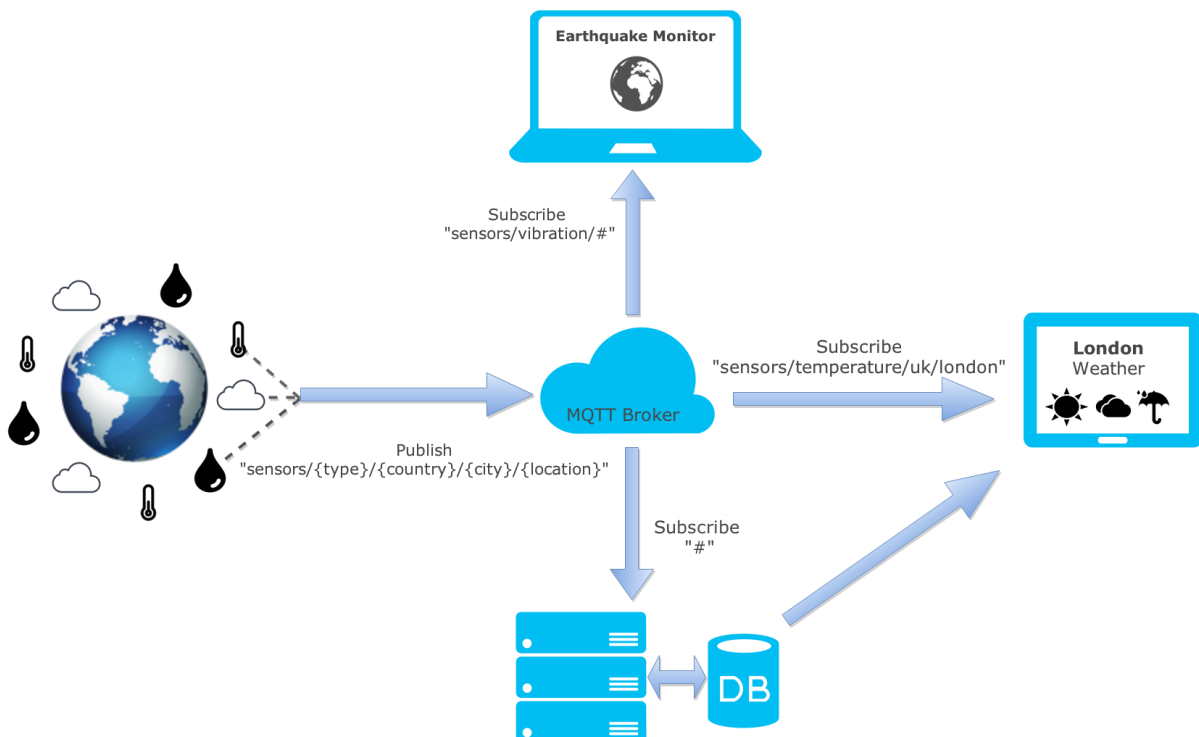


Figure 10 MQTT Wildcard example setup [18]

As depicted in figure 11, MQTT is a modular system. Adding new sensors and databases could be considered a simple task. MQTT brokers can be highly parallelized and event-driven making a single broker easily scalable to tens of thousands of messages per second. [18]

### 3.2.7 Summary

MQTT is a publish/subscribe protocol running on top of TCP and was originally developed in 1999. In 2013, it was turned over to the OASIS organization. The current OASIS standard version of MQTT is 3.1.1 and this version was approved on the 29th of October 2014. [14]

**Ideal for constrained networks:** MQTT control packet headers are kept as small as possible. Each MQTT control packet consist of three parts, a fixed header, variable header and payload. Each MQTT control packet has a 2-byte Fixed header. Not all the control packets have the variable headers and payload. A variable header contains the packet identifier if used by the control packet. A payload up to 256 MB could be attached in the packets. Having a small header overhead makes this protocol appropriate for IoT by lowering the amount of data transmitted over constrained networks. [14]

**Reliable:** The MQTT protocol allows messages to be exchanged with a range of QoS, from “At most once” to “Exactly-once” acknowledged delivery. [14]

**Flexible:** MQTT is open protocol and standardized by OASIS. This makes this protocol easy to adopt for the wide variety of IoT devices, platforms, and operating systems. Many applications of MQTT can be developed just by implementing the CONNECT, PUBLISH, SUBSCRIBE, and DISCONNECT control packets. [14]

## 3.3 AMQP

### 3.3.1 Overview

AMQP is a protocol that like MQTT, communicates via publish/subscribe and operates on top of TCP. It was originally developed in 2003 by John O’Hara at JPMorgan Chase and iMatix with the aim of creating an interoperable message system that was non-proprietary and could be used as a standard messaging protocol for investment banks. AMQP allows vendor-specific and standards-agreed future extensions in a way compatible with, and usable by, existing implementations. The protocol of AMQP is layered, allowing change in one part of the specification to be isolated from another.

AMQP was announced to be incorporated in OPC-UA as late as 6<sup>th</sup> of April, 2016. [19]

However, AMQP was not designed to have a small code footprint or an easy to use interface, AMQP was rather designed to be feature rich and high performance. Additionally, it is not simply a messaging protocol, but also defines its own type system to ensure interoperability between client and server. This is a major difference from MQTT, where an important factor of the protocol is simplicity. [20]

The main benefit of AMQP is its robust communication model that supports transactions. Unlike MQTT, AMQP can guarantee completion of transactions, which, though useful, is not always required by IoT applications. AMQP often gets grouped with IoT protocols and it is one. But its biggest con is that it is a heavy protocol. It was initially meant for backend IT systems, and not the edge of the network. [21]

Since the AMQP protocol is exhaustive, this thesis tries to cover only the most important and central components of AMQP.

- Exchange

A part of the broker which receives messages and routes them to queues.

- Queue (Message queue)

A named entity which messages are associated with and from where consumers/subscribers receive them. Messages wait here until read.

- Bindings

Rules for distributing messages from exchanges to queues.

### 3.3.2 Communication

A publisher sends messages to a named exchange and a consumer pulls messages from a queue (or the queue pushes them to the consumer depending on the configuration). For this to work, the connections have to be made already. This connection is made via the name of the exchange. Usually, either the publisher or consumer creates the exchange with a given name and then makes that name public. How that publication happens depends on the circumstances and implementation (see chapter 4.3.2 for the method used in this thesis), but methods such as presenting it in a public API documentation or send it to known clients could be used.

The next step is the transfer/routing of messages from the exchange to the queue. Initially, the queue has to be attached to the given exchange. Typically, a consumer/subscriber creates a queue and attaches it to an exchange at the same time. Messages received by the exchange have to be matched to the queue - a process mentioned above called binding. [22]

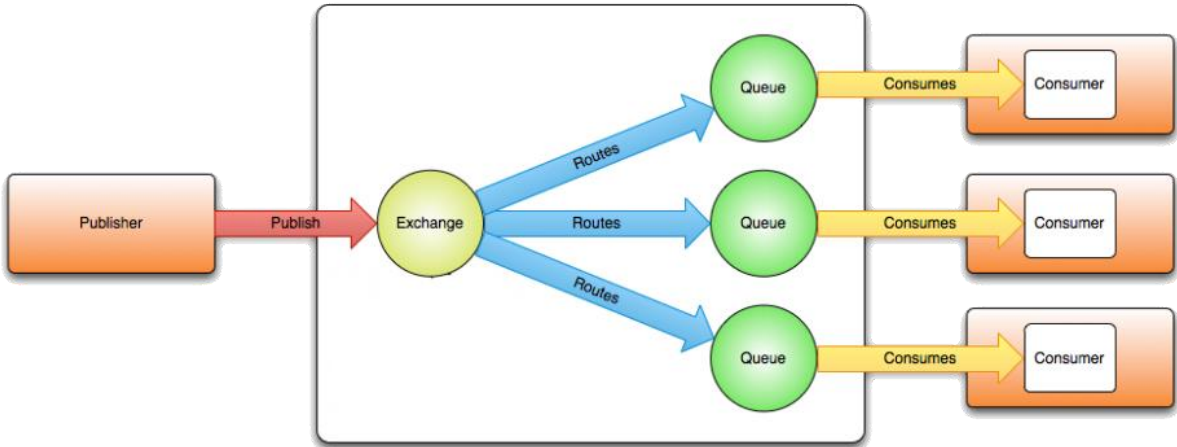


Figure 11 AMQP Communication flow

The communication from the publishers to exchanges and from queues to subscribers uses TCP. Further, endpoints must acknowledge acceptance of each message. The AMQP standard also describes an optional transaction mode with a formal multiphase commit sequence. Due to its origins in the banking industry, AMQP middleware focuses on tracking all messages and ensuring each is in fact delivered as intended, regardless of failures or reboots. [22]

#### 3.3.2.1 Bindings

Before looking into bindings, a AMQP message has the following format: the first part consists of headers, the second part properties and the third part consists of a byte array.

The headers and properties of the message are basically key/value pairs (hash-table). The difference between them is that the headers are defined by the AMQP specification whereas properties can contain arbitrary, application-specific information. The actual message content is simply a sequence of bytes.

One of the standard headers is called routing-key and it is this that the broker uses to match messages to queues. Each queue specifies a “binding key” and if this key matches the value of the routing-key header, the queue receives the message. [22]

#### 3.3.3 Quality of service

AMQP implements the same QoS levels as MQTT. Namely “Exactly-Once”, “At least once” and “At most once”. When the message exchange starts, the sender assigns a delivery tag to the message in order to track its delivery. Both peers have a map with settlement status of the messages in transit and each message starts with an unsettled status. [22]

#### 3.3.4 Last will

AMQP lacks the last-will feature. [22]

#### 3.3.5 Security

Security layers are used (externally to the AMQP protocol) to establish an authenticated and/or encrypted transport over which regular AMQP traffic can be tunneled. AMQP supports the SASL security layer which depends on its host protocol to provide framing. [22]

#### 3.3.6 Wildcards

AMQP supports wildcards, this is done via the different exchange types:

Exchange type	Description
---------------	-------------

Direct	The binding key must match the routing key exactly - no wildcard support.
Topic	Same as Direct, but wildcards are allowed in the binding key. # matches zero or more dot-delimited words and * matches exactly one such word.
Fanout	The routing and binding keys are ignored - all published messages go to all bound queues.

Table 5 AMQP Exchange types

### 3.3.7 Summary

AMQP is an OASIS standard with the goal of defining the mechanics of the secure, reliable, and efficient transfer of messages between two parties. The messages themselves are encoded using a portable data representation that enables heterogeneous senders and receivers to exchange structured business messages at full fidelity. The following is a summary of the most important features: [22]

**Efficient:** AMQP is a connection-oriented protocol that uses a binary encoding for the protocol instructions and the business messages transferred over it. It incorporates sophisticated flow-control schemes to maximize the utilization of the network and the connected components. That said, the protocol was designed to strike a balance between efficiency, flexibility and interoperability. [22]

**Reliable:** The AMQP protocol allows messages to be exchanged with a range of QoS, from “At most once” to “Exactly-once” acknowledged delivery. [22]

**Flexible:** AMQP is a flexible and can be used to support different topologies. The same protocol can be used for client-to-client, client-to-broker, and broker-to-broker communications. [22]

**Broker-model independent:** The AMQP specification does not make any requirements on the messaging model used by a broker. This means that it is possible to easily add AMQP support to existing messaging brokers. [22]

## 3.4 OPC-UA

### 3.4.1 Overview

OPC-UA is the next generation of OPC technology. OPC-UA is more secure, open and reliable mechanism for transferring information between servers and clients. It provides more open transports, better security and a more complete information model than OPC (Classic, discussed in next section). It provides an effective method to move data between enterprise systems and sensors, actuators, PLCs and other devices that interacts with real world processes. [23]

To relate to the classical communication pyramid, OPC-UA can be used between all layers of the pyramid (with the exception of the component layer, in the component layer there may however be OPC-UA compatible components but not in the general case). [23]

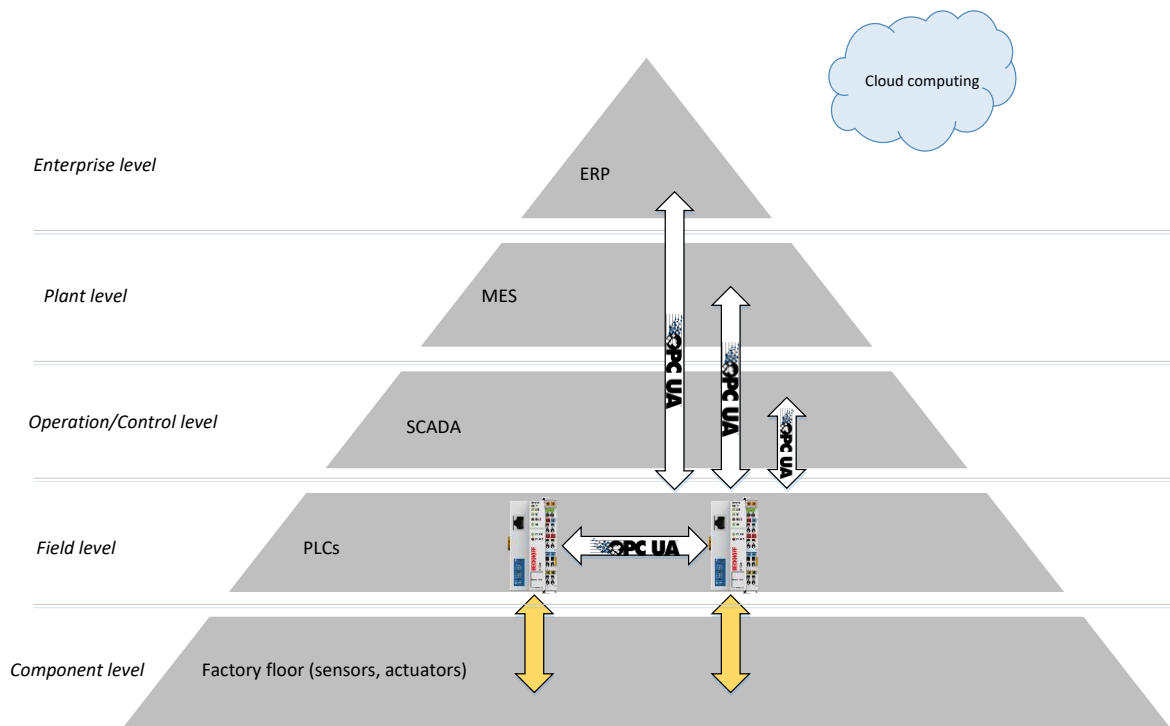


Figure 11 OPC-UA in the classical communication pyramid

OPC-UA specifies a client-server architecture, where a client can perform several operations on the server's address space. Such operations are access, read and even modify the server's address space.

Before digging deeper into the OPC-UA technology and how it can be used to send data to the cloud, we need to understand the difference and benefits comparing to the OPC (also known as OPC Classic) technology. In the next section a small overview about OPC Classic is therefore given. [23]

### 3.4.2 OPC Classic

OPC was formerly an acronym for "OLE (Object Linking and Embedding) for Process Control", however this acronym has been changed to "Open Platforms Communication", in order to be more close to the purpose of OPC-UA, rather than OPC classic. The motivation for OPC Classic was however straightforward: a (machine) vendor would implement an OPC server on their respective devices. Any OPC client could then simply read values (tag values) from the server, and in turn send those values to higher-level systems such as ERP, MES and SCADA/HMI. An OPC Server therefore filled a very important role in the IT infrastructure of a factory. It was a well-defined server where clients could connect and fetch process data needed for upper-level systems.

OPC Classic is built around DCOM (Distributed Component Object Model) from Microsoft. DCOM is a technology that is obsolete and de-emphasized by Microsoft. The dependency of DCOM made OPC Classic totally dependent on Microsoft platforms. In OPC Classic there are no built-in functionality to model data adequately. It completely lacks the ability to define relationships of data and information hosted by the server. This is a major improvement in OPC-UA where a server can build complex information models. [23]

### 3.4.3 The address space

The OPC-UA address space model begins with a base element – that is, a node. A node is a structured data element consisting of a set of predefined attributes and relationships to other nodes. Nodes are simply connected to each other by having one or more References to the node they relate to. Each node has a set of attributes, and a set of References. [23]

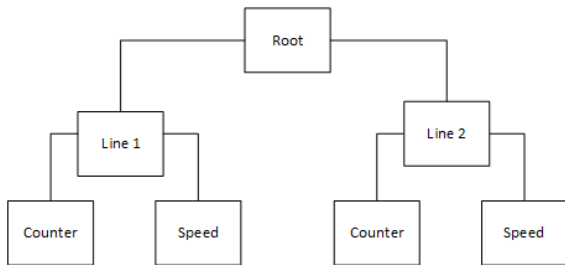


Figure 12 Simple OPC-UA address space

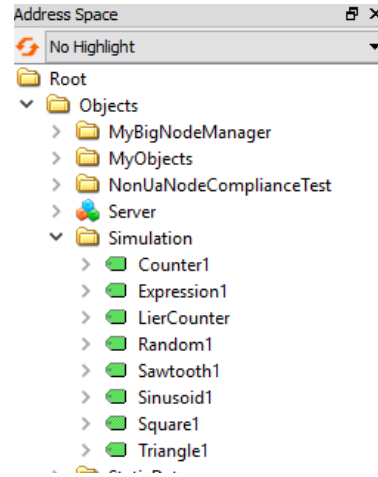


Figure 13 Address space as seen in commercial OPC-UA client “UaExpert” from Unified Automation.

The OPC-UA address space is designed hierarchically, and all top-level nodes are the same in all OPC-UA servers in order to promote interoperability.

Attributes are elementary components of a node; the attribute definition consists of the following information:

- Attribute ID
- Name
- Description
- Data type
- A mandatory/optional indicator

[23]

### 3.4.4 Services

A OPC-UA server provides ten standard services [23]:

Service set	Description
Discovery	The Discovery Service Set provides the services a client uses to discover connection endpoints in the server and evaluate the capability of the endpoints to meet the client application requirements. Each server has a discovery endpoint that the clients can access without establishing a session.

Secure channel	The Secure Channel Service Set provides services to establish and maintain secure links between client and server. When a client has been authenticated, the Secure Channel Service Set creates a long-term communication channel that maintains the confidentiality and integrity of messages sent between client and server.
Session	The Session Service Set establishes a communication session between client and server. The client can pass different credentials to the server: a user name and a password, an X.509 certificate, a WS-token or a static string "anonymous". The Server needs to be configured to accept anonymous login.
Node management	The Node Management Service Set provides services to create and delete nodes in the server's address space.
View	The View Service Set enables the possibility to partition the Address Space into subsets.
Query	The Query Service Set provides functionality to get bulk data access. This is especially interesting when working with large historical data for instance.
Attribute	The Attribute Service Set provides functionality to read and write attributes of nodes in the Address Space.
Method	The Method Service Set is used for the client to invoke methods. Methods are logic programs that the OPC-UA servers makes available. Parameters can also be passed (defined as properties of the method).
Monitored item	The Monitored Item Service Set provides important functionality to be able to create, modify and delete Monitored Items. Monitored Items generates notifications on data changes. Data changes can be changes in node attributes or node values. A change will trigger the server to send a notification. Alarms and events can also be used to trigger notifications.
Subscription	The Subscription Service Set provides functionality to manage subscriptions on the server. A subscription has one or many monitored items and tells the server what publishing interval shall be used etc.

Table 6 OPC-UA standard services

## Chapter 4: Method

### 4.1 Introduction

This chapter will describe the process of designing, evaluating and implementing the tests and concepts related to transferring data from an industrial factory floor to an enterprise-level system.

## 4.2 Pre-Study

The process was performed in several stages, the first stage was a pre-study. The pre-study included an exhaustive study of what several PLC vendors are offering as IoT solutions, what technologies they are using and what their devices are capable of. The pre-study also included meeting architects of PLC vendors such as Beckhoff – in order to get a picture of their roadmaps and ideas of how they are planning to implement IoT solutions in the future.

Many PLC vendors are – as of writing November 2016 – not offering devices that can communicate directly with MQTT or AMQP. However, this is something that is in the roadmap of several PLC vendors.

Due to the lack of devices to test the communication protocols, the tests in this thesis were carried out on hardware similar to some industrial PLC already released and upcoming. The Raspberry Pi 3 will be used to simulate a PLC.

A goal of this thesis is to evaluate and research about what possibilities there are to send industrial data from the factory floor to an enterprise level software with the two different protocols and their pros and cons. This goal becomes quickly extremely implementation and vendor specific. For instance, the Azure IoT libraries does not – as of writing – support MQTT when communicating with their IoT hub. Microsoft states *“The Universal Windows Platform (UWP) version of the .NET client device library does not currently support MQTT protocol.”* [24]. This is serious limitation found during the evaluation. However, while most implementation such as this will be fixed in time, the more important focus is the actual concept and how the components will interact with each other, rather exactly what device or programming language is used.

## 4.3 Implementation

Two test benches and proof-of-concepts were set up. One for MQTT and one for AMQP. The first part of the MQTT test bench are built upon a custom signal generator, MQTT library and publisher software. The middle point is an Azure Virtual Machine running a MQTT broker software. And the endpoint consisted of a subscriber software to collect the data.

The AMQP test bench is built upon OPC-UA. A commercial OPC-UA server was installed (on a regular Windows Server) and configured to generate a ramp function. The OPC-UA server was installed on a regular Windows Server due to simplicity. In the general case, the OPC-UA server might have been located on the PLC software itself. The first part consists of a publisher application embedded within the OPC-UA stack. The second part, similar to the MQTT test bench a subscriber software was used as an endpoint.

### 4.3.1 MQTT

As stated, the UWP (see section below) IoT Azure libraries do not support connecting to an Azure IoT Hub with the MQTT protocol. This limitation obviously means that different brokers will be used for MQTT and AMQP. Another limitation is that the Azure IoT Hub does not support Quality-of-Service level 2 (as of January 2017). However, the implementation was made by using an Azure Virtual Machine on which a third-party MQTT broker called Mosquitto was installed. The test schema looks as following:

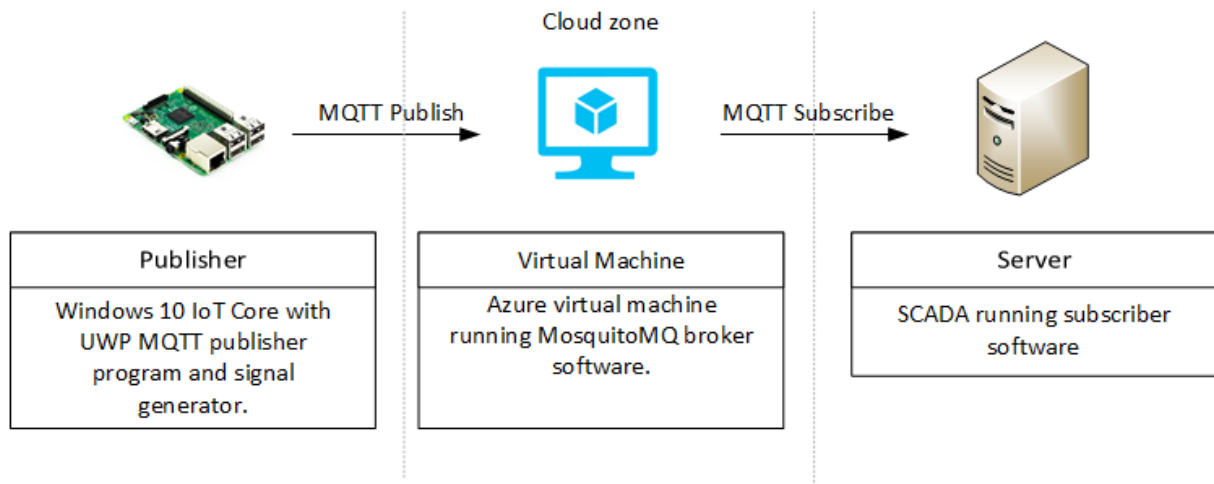


Figure 14 MQTT test bench setup

The code for the left part “Publisher” of this test bench is available in the appendix with the title “MQTT Publisher software”. The “Server” part is available under the title “MQTT Subscriber”.

The MQTT implementation starts by connecting to the broker by sending a CONNECT packet. When the connection is successful, the publisher continues to publish messages with different QoS in a sequence. The latency is measured in the publisher’s code. The publisher uses a hardcoded topic for measurement.

#### 4.3.2 AMQP

The OPC-UA stack and examples are used and extracted to test AMQP. The first step for this test is to perform a full-stack test with an out-of-the-box configuration, which means from data change on the OPC-UA server to message received on the subscriber program. Prosys Simulation server was used as OPC-UA server, the server has been configured to host a few OPC-UA nodes (ramp function) that output integer values.

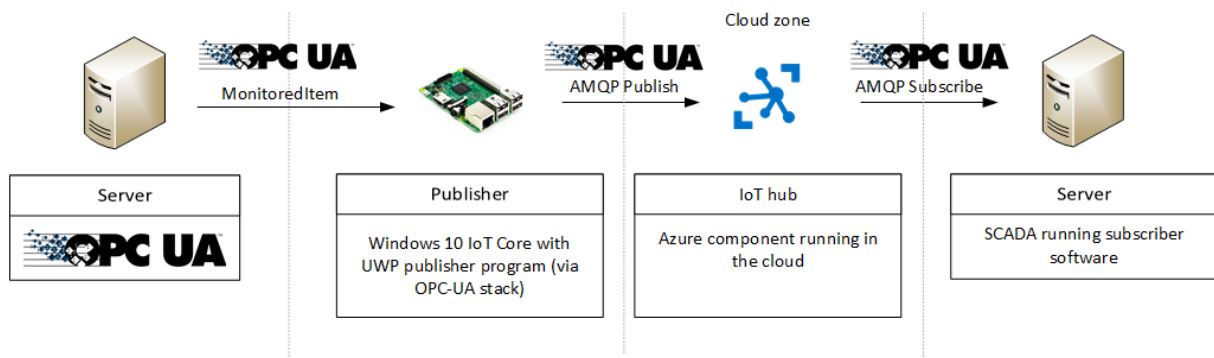


Figure 15 AMQP/OPCUA test bench setup

The relevant code for the left part “Publisher” of this test bench is available in the appendix with the title “AMQP Publisher method”. The “Server” part is available under the title “AMQP Subscriber”.

The AMQP implementation starts by setting up a device (Microsoft Azure term used to identify any device that will send events to an IoT Hub). The AMQP then publishes data in similar manner to the MQTT implementation. The latency is measured on client side and with a static IoT hub URI. Since the publisher uses the Azure API, the AMQP exchange are pre-configured.

### 4.3.3 Raspberry Pi

The publisher software was deployed on a Raspberry Pi. In a production environment, the publisher software would run on a PLC. Therefore, a comparison with a PLC from Beckhoff is made:

Device	Raspberry Pi 3	Beckhoff CX 9020
Architecture	ARMv8	ARMv7-A
Processor	ARM Cortex-A53, 1.2GHz QuadCore	ARM Cortex™-A8, 1 GHz
RAM	1 GB	1 GB
OS	Microsoft Windows 10 IoT Core	Microsoft Windows Embedded Compact 7

Table 7 Test bench hardware comparison with PLC

While both devices are obviously not identical in hardware, the tests should give a rough estimation of what performance would be expected from an industrial-ready PLC. However as stated, the focus of the project is rather slower paced data streams and not M2M communication. Therefore, performance is not the most important factor when choosing protocols, but rather other factors such as flexibility, security and availability.

### 4.3.4 Windows 10 IoT Core

In order to have a solid development environment, Windows 10 IoT Core was installed on the Raspberry Pi. This enables the development of UWP with C# code. As of November 2016, both libraries used for the tests for this thesis are supported to run on the UWP architecture. The Microsoft OPC-UA also support a large part of the OPC-UA stack to run in UWP.

Windows 10 IoT Core was chosen in order to try the technologies with the latest cutting-edge technologies. Windows 10 IoT Core was first released to the public in April 2015. [25]

### 4.3.5 Software

In this section the software and technologies used for the tests are presented.

#### 4.3.5.1 UWP

Windows 8 introduced the Windows Runtime (WinRT), which was an evolution of the Windows app model. It was intended to be a common application architecture.

Microsoft has undergone a major transformation that changed the way developers write programs, and use the development tools, and most importantly the way they interact with the underlying runtime environment. The result is the convergence of a program model, which in effect created a platform where the operating system maintains a set of APIs across all sets of hardware platforms while still providing flexibility by way of extensions on a specific set of hardware platforms called Device Family. UWP (Universal Windows Platform) is the result of a combined and unified core.

#### 4.3.5.2 *Signal simulation*

For the MQTT, the values will be simulated as if they were outputted by the PLC. In the AMQP test, data from a OPCUA server will be used directly. However, the underlying signals will be simulated in the OPC-UA server by configuration of a ramp function.

#### 4.3.5.3 *Mosquitto*

Eclipse Mosquitto is an open source message broker that implements the MQTT protocol versions 3.1 and 3.1.1. This software runs in an Azure Virtual Machine for the test presented in this thesis.

#### 4.3.5.4 *M2Mqtt*

M2Mqtt is an open-source MQTT implementation library available as a NuGet package. The library was created by Microsoft employee Paolo Patierno. The library is supported on a wide range of .NET platforms as well as on the UWP architecture.

#### 4.3.5.5 *Azure IoT SDK*

As stated, AMQP is very recently being included in the OPC-UA specification. As of time of writing, the specification for OPC-UA publish/subscribe mechanism is not yet released. However, the reference stack includes all technologies and source code for the implementation. These applications are still not – as of writing in December 2016– ready to be used directly against the Azure IoT Hub. Therefore, the Azure IoT SDK can be used with identical result. The Azure IoT SDK uses QoS=1, therefore results for this quality of service level is only available in the results.

## 4.4 Evaluation

The last step was to extract all data from the implemented test benches and software. This data was carefully studied in order to mitigate faults introduced from implementation specific details. The data was also studied carefully with respect to plausibility. Manual debugging of protocol features was also carefully done in order to make sure no extra paths in the protocol codes were executed. Mitigation of faults also included test benches implemented in different environments with respects to network and firewalls. Finally, the data results were linked to the theory and studied.

# Chapter 5: Results

## 5.1 Introduction

This chapter describes the tests' results.

## 5.2 MQTT

The following result is based on a typical value message from a PLC, i.e. a 32-bit integer value and metadata (timestamp, message ID etc), resulting in less than 100 bytes. The latency is based on the time needed to publish the data to the broker.

5.2.1 Major results

Description	Result
Average latency (QoS = 2)	108 [ms]
Average latency (QoS = 1)	51 [ms]

Table 8 MQTT major results

In the following graph, latency is measured for MQTT with QoS 2 and QoS 1 with respect to an increasing payload size. (The maximum payload size test is 0x40000 (262144) due to Azure’s payload limitation).

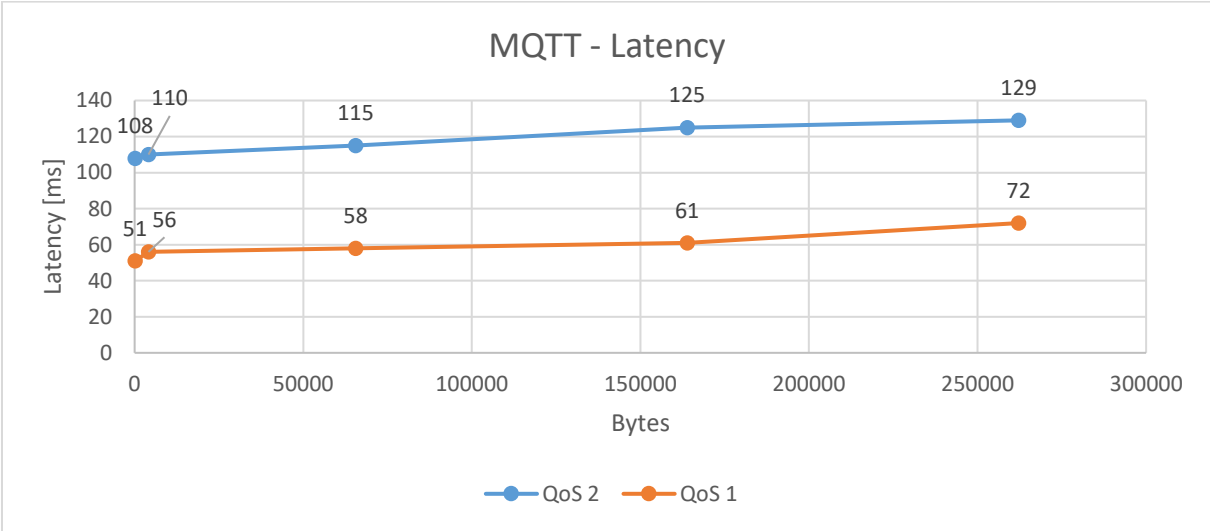


Figure 16 MQTT Latency measurement results

5.3 AMQP

5.3.1 Major results

The following latency is based on a typical value message from a PLC, i.e. a 32-bit integer value and metadata (timestamp, message ID etc.), resulting in less than 100 bytes.

Description	Result
Average latency (QoS 1)	113 [ms]

Table 9 AMQP major results

In the graph below, the AMQP latency is measured with respect to an increasing message size.

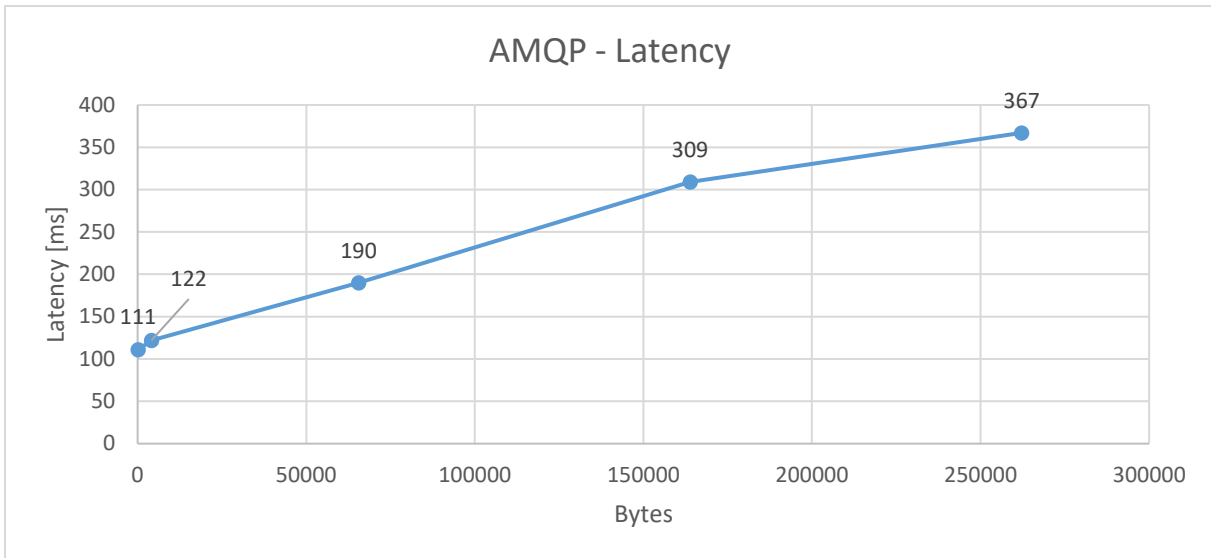


Figure 17 AMQP Latency measurement results

## 5.4 Latency comparison

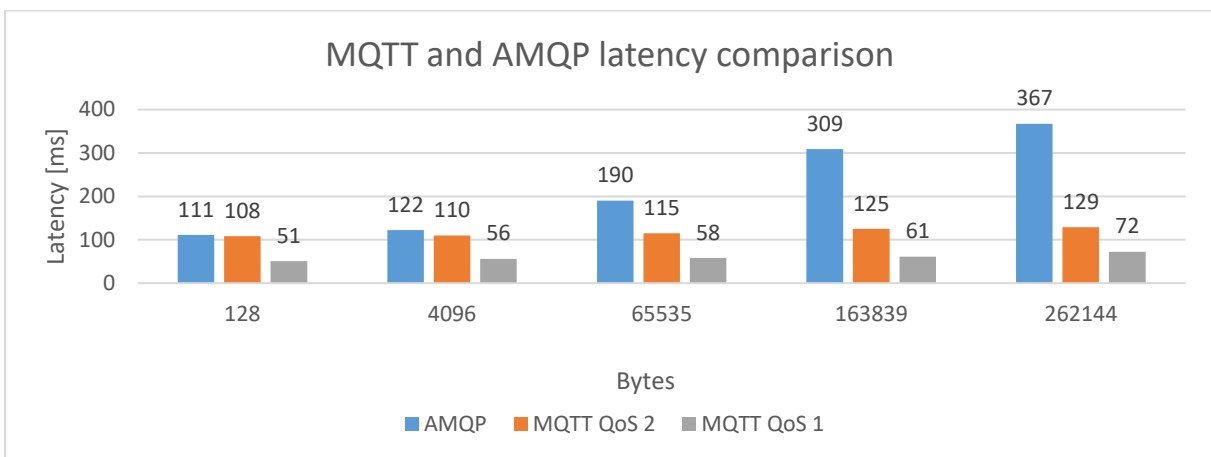


Figure 18 MQTT and AMQP Latency comparison

## 5.5 Features

The functionalities and features of AMQP and MQTT have also been evaluated, the security aspects show that AMQP has more advanced built-in support for more exhaustive security. While MQTT relies mostly on implementation specific security. Results also show that both AMQP and MQTT can handle wildcards in a very similar manner.

MQTT and AMQP are both suitable for use in hardware and software and on all major operating systems and platforms. MQTT is suited to its use case of simple clients talking to a server, but any infrastructure using it is exposed to serious security weaknesses and an inability to make best use of resources or to support additional use cases. AMQP is ready to use out-of-the-box in public cloud systems such as Azure, not only from traditional x86 systems but also from UWP.

AMQP is suited to these uses cases and many others, supports far better use of resources, far more pragmatic security and message reliability and has a future place as an ISO standard.

## Chapter 6: Discussion

Here the results presented in the previous chapter will be analyzed and discussed in an informal manner.

In the previous chapter, we have seen how factory floor signals can be transferred all the way to top level systems. Initially, the background indicated that MQTT would give a lower latency, and as the result shows: this is also the case. MQTT (with standard payloads) with the Quality-of-Service Exactly once resulted in virtually the same latency as AMQP with At-least-once quality of service. This is however not valid for relatively huge payloads (e.g. over 65535 as seen in the results). Further, with respect to industrial communication when transferring relatively small payloads to a MES, the latencies of the two protocols are virtually identical.

As seen in the results, no measurements are naturally present for Quality-of-Service At most once (0). This is due to the fact that no acknowledgement from the broker is sent and this enables MQTT to send data streams below the Quality-of-Service “At least once” pace but with very limited or no reliability.

As seen in the background chapter, the use-cases of AMQP have taken a different direction that AMQP was initially designed for. It is very interesting that OPC Foundation decided to use AMQP instead of MQTT or other IoT protocol for their standard OPC-UA specification. OPC-UA will most likely use QoS At-least-once, one could argue that this QoS is “good enough” and potential redundancy of messages will be handled with external software, even within the OPC-UA protocol itself. The results in this thesis show that OPC-UA is unlikely to suffer from latency issues due to its choice of IoT protocol. This is an important aspect because it enables OPC-UA to use the richer AMQP feature-set. The AMQP protocol has a steeper learning curve however, this is in the benefit of MQTT for standalone implementations. However, when AMQP is embedded in OPC-UA, this will be a minor issue.

We have also seen how it is possible to implement the state-of-the-art OPC-UA stack on Microsoft’s newest technologies such as IoT Core and UWP. It is probable that many PLCs will in the future use at least IoT Core as their host OS. Public cloud system has also been proven to be ready for industrial usage all the way from the production floor to an ERP system in the cloud.

## Chapter 7: Conclusions

With the advance of IIoT and Industry 4.0, interoperability between industrial systems becomes more vital than ever before. Likewise, these concepts are important when the realization of innovative new business models is a requirement for the underlying infrastructure. This also drives the increased convergence of IT and automation technologies. Cloud-based data services can help implement such automation projects, as they save the machine manufacturer or end customer from having to provide the corresponding IT expertise.

The possibility to exchange data in the fastest and easiest way possible is an important requirement when we talk about Industry 4.0. In fact, it is based on the concept of efficiency and Internet of Things, for which automatic communication among devices is necessary.

Many industries have already started the transitions to the private and public cloud. It has also been seen how many ERP vendors publish interfaces using protocols such as MQTT and OPC-UA. Thus,

enabling these protocols as connectors in the Industry 4.0 transition. Naturally, this allows a system to collect and organize any kind of data coming from various devices.

This new movement is not only developing in the manufacturing sector, but as well in many other sectors. In manufacturing, smart factories with automated maintenance are becoming a reality, bringing major improvements to efficiency.

In the financial services sector, automation is being used to cope with an ever-increasing volume of data, enhancing customer service and enabling more time to be devoted to areas such as security and risk.

## 7.1 Future work

This thesis has been limited to MQTT and AMQP. As stated, several other IoT protocols exist and should be investigated for possible usage. With the respect to PLC vendors, many vendors will start supplying IoT-ready devices, i.e. PLCs with IoT-ready blocks that support protocols such as MQTT and AMQP. The performance and scalability of these implementations will be crucial for the device's IoT future.

The thesis has also been limited by using only a single PLC. A more exhaustive analysis and next step would be to include several hundreds of units simultaneously in order to test more performance heavy scenarios.

However, it is unlikely that the protocols themselves will have an impact, but rather the network and the implementation.

## Chapter 8: Bibliography

- [1] R. Elliott, "Manufacturing Execution System (MES) An Examination of Implementation Strategy," Faculty of California Polytechnic State University, 2013.
- [2] S. Karnouskos and A. W. Colombo, "Architecting the next generation of service-based SCADA/DCS system of systems," SAP Research, 2011.
- [3] D. H. Sheldon, "Class A ERP Implementation," J. Ross Publishing, Incorporated, 2005.
- [4] K. Stouffer, V. Pillitteri, S. Lightman, M. Abrams and A. Hahn, "Guide to Industrial Control Systems (ICS) Security," NIST.
- [5] D. Kolberg and D. Zühlke, "Lean Automation enabled by Industry 4.0 Technologies," ScienceDirect, Kaiserslautern.
- [6] J. Pfrommer, "OPC UA & Industrie 4.0 - enabling technology," Karlsruhe Institute of Technology, 2016.
- [7] L. Gehrke, A. T. Kühn, D. Rule, P. Moore, C. Bellmann, S. Siemes, D. Dawood, S. Lakshmi, J. Kulik and M. Standley, "A Discussion of Qualifications and Skills in the Factory of the Future: A German and American Perspective," ResearchGate, Düsseldorf, 2015.
- [8] "Gartner Says the Internet of Things Will Transform the Data Center," 19 March 2014. [Online]. Available: <http://www.gartner.com/newsroom/id/2684616>. [Accessed 12 November 2016].
- [9] K. Lee and I. Lee, "The Internet of Things (IoT): Applications, investments, and challenges for enterprises," Science Direct, 2016. <http://www.sciencedirect.com/science/article/pii/S0007681315000373> [Accessed 26 November 2016].
- [10] K. Klawon, J. Gold, K. Bachman and D. Landoll, "Considering IIOT and Security for the DoD," University of Dayton Research Institute. <http://proceedings.spiedigitallibrary.org/proceeding.aspx?articleid=2523441> [Accessed 25 November 2016].
- [11] M. A. Prada, P. Reguera, S. Alonso, A. Moráan, J. J. Fuertes and M. Domínguez, "Communication with resource-constrained devices through MQTT for control education," *Science Direct*, p. 2, 2016.
- [12] "MQTT version 3.1.1 becomes an OASIS standard," OASIS, 29 October 2016. [Online]. Available: <https://www.oasis-open.org/news/announcements/mqtt-version-3-1-1-becomes-an-oasis-standard>. [Accessed 28 November 2016].
- [13] R. Gupta, "5 Things to Know About MQTT," IBM, 23 September 2014. [Online]. Available: [https://www.ibm.com/developerworks/community/blogs/5things/entry/5\\_things\\_to\\_know\\_about\\_mqtt\\_the\\_protocol\\_for\\_internet\\_of\\_things?lang=en](https://www.ibm.com/developerworks/community/blogs/5things/entry/5_things_to_know_about_mqtt_the_protocol_for_internet_of_things?lang=en). [Accessed 24 November 2016].

- [14] "MQTT Version 3.1.1.," OASIS, 10 December 2015. [Online]. Available: <http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html>. [Accessed 28 November 2016].
- [15] "MQTT Essentials Part 3: Client, Broker and Connection Establishment," HiveMQ, [Online]. Available: <http://www.hivemq.com/blog/mqtt-essentials-part-3-client-broker-connection-establishment>. [Accessed 12 November 2016].
- [16] "MQTT Essentials Part 4: MQTT Publish, Subscribe & Unsubscribe," HiveMQ, [Online]. Available: <http://www.hivemq.com/blog/mqtt-essentials-part-4-mqtt-publish-subscribe-unsubscribe>. [Accessed 10 November 2016].
- [17] M. H. Amaran, N. A. M. Noh, M. S. Rohmad and H. Hashim, "A Comparison of Lightweight Communication Protocols in Robotic Applications," *Procedia*, 2015.
- [18] B. Howes, "A Brief, but Practical Introduction to the MQTT Protocol and its Application to IoT," *Zoetrope*, 23 March 2016. [Online]. Available: <https://zoetrope.io/tech-blog/brief-practical-introduction-mqtt-protocol-and-its-application-iot>. [Accessed 8 December 2016].
- [19] O. Foundation, "OPC Foundation Announces support of Publish / Subscribe for OPC UA," 6 April 2016. [Online]. Available: <https://opcfoundation.org/news/opc-foundation-news/opc-foundation-announces-support-of-publish-subscribe-for-opc-ua/>. [Accessed 30 November 2016].
- [20] J. O'Hara, "Toward a Commodity Enterprise Middleware," 2007.
- [21] A. Semle, "IIoT Protocols to Watch," 2015. [Online]. Available: [http://www.automation.com/pdf\\_articles/kepware/IIoT\\_Protocols\\_to\\_Watch.pdf](http://www.automation.com/pdf_articles/kepware/IIoT_Protocols_to_Watch.pdf). [Accessed 25 November 2016].
- [22] "AMQP 1.0 specification," OASIS, 29 October 2012. [Online]. Available: <http://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf>. [Accessed 28 November 2016].
- [23] J. S. Rinaldi, *OPC UA - Unified Architecture: The Everyman's Guide to the Most Important Information Technology in Industrial Automation*, Createspace Independent Publishing Platform, 2016.
- [24] "Microsoft Azure IoT device SDK FAQ," Microsoft, 11 October 2016. [Online]. Available: <https://github.com/Azure/azure-iot-sdks/blob/master/doc/faq.md>. [Accessed 29 November 2016].
- [25] S. Teixeira, "Microsoft brings Windows 10 to Makers," Microsoft, 29 April 2015. [Online]. Available: <http://blogs.windows.com/buildingapps/2015/04/29/microsoft-brings-windows-10-to-makers/>. [Accessed 30 November 2016].
- [26] S. G. McCrady, *Designing SCADA application software*, London: Waltham, Mass., 2013.
- [27] J. Kletti, *Manufacturing execution systems*, Berlin ; New York: Springer, 2007.
- [28] W. Bolton, *Programmable logic controllers*, Kidlington, Oxford, 2015.

# Appendix

## MQTT Publisher software

```
using Newtonsoft.Json;
using System;
using System.Collections.Generic;
using System.Diagnostics;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

namespace AxIIoT.Mqtt.Publisher
{
    public class Service
    {
        public Service()
        {
            // MQTT Connection settings
            string mqttHost = "axzapp06";

            // MQTT Publish Settings
            string mqttTopic = "machineline100/productioncounter";
            byte mqttQosLevel = MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE;
            bool mqttRetain = false;

            // Publishing timing settings
            int productionCycleTimeInSeconds = 2;
            int productionStopAfterCycles = 10;
            int productionStopIntervalInSeconds = 10;

            // create MQTT client to connect to broker hosted in Azure Virtual Machine
            MqttClient client = new MqttClient(mqttHost);

            // connect to the broker
            client.Connect(Guid.NewGuid().ToString());

            client.MqttMsgPublished += Client_MqttMsgPublished;

            // create the signal generator
            SignalGenerator generator = new RampFunction();

            int productionCycleCounter = 0;

            int qosCounter = 0;

            // production loop
            while(true)
            {
                // publish data to the cloud
                m_publishTimer = new Stopwatch();
```

```

        if (qosCounter >= 0 && qosCounter <= 10)
        {
            mqttQosLevel = MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE;
        }
        else if (qosCounter >= 11 && qosCounter <= 20)
        {
            mqttQosLevel = MqttMsgBase.QOS_LEVEL_AT_LEAST_ONCE;
        }
        else if (qosCounter >= 21 && qosCounter <= 30)
        {
            mqttQosLevel = MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE;
        }

        var telemetryDataPoint = new
        {
            deviceId = "emain2:" + mqttQosLevel.ToString(),
            previousPublishTime = m_previousPublishTime,
            timestamp = DateTime.Now.ToString(),
            counter = generator.Get()
        };
        var messageString = JsonConvert.SerializeObject(telemetryDataPoint);

        m_publishTimer.Start();
        client.Publish(
            mqttTopic,
            Encoding.ASCII.GetBytes(messageString),
            mqttQosLevel,
            mqttRetain);

        // simulate production cycle
        System.Threading.Tasks.Task.Delay(
            productionCycleTimeInSeconds * 1000).Wait();

        // simulate production downtime
        if (productionCycleCounter >= productionStopAfterCycles)
        {
            productionCycleCounter = 0;
            System.Threading.Tasks.Task.Delay(
                productionStopIntervalInSeconds * 1000).Wait();
        }

        qosCounter++;
    }
}

e) private void Client_MqttMsgPublished(object sender, MqttMsgPublishedEventArgs
    {
        m_publishTimer.Stop();
        m_previousPublishTime = m_publishTimer.Elapsed.ToString();
    }

    private Stopwatch m_publishTimer;
    private string m_previousPublishTime = "0";
}
}

```

## RampFunction.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AxIIoT.Mqtt.Publisher
{
    public class RampFunction : SignalGenerator
    {
        #region Constructors
        public RampFunction()
        {
            Initialize();
        }

        private void Initialize()
        {
            // init value
            m_currentValue = 0;
        }
        #endregion

        public override int Get()
        {
            // iterate to next value
            NextValue();
            // return value
            return m_currentValue;
        }

        private void NextValue()
        {
            m_currentValue++;
        }

        #region Private Fields
        private int m_currentValue;
        #endregion
    }
}
```

## SignalGenerator.cs

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

namespace AxIIoT.Mqtt.Publisher
{
    public abstract class SignalGenerator
    {
        public abstract int Get();
    }
}
```

## MQTT Subscriber

```
using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;
using uPLibrary.Networking.M2Mqtt;
using uPLibrary.Networking.M2Mqtt.Messages;

namespace AxIIoT.Mqtt.Subscriber
{
    class Program
    {
        static void Main(string[] args)
        {
            MqttClient client = new MqttClient("axzapp06");
            client.MqttMsgPublishReceived += Client_MqttMsgPublishReceived;
            client.MqttMsgPublished += Client_MqttMsgPublished;
            client.Connect(Guid.NewGuid().ToString());

            string[] topic = { "machineline100/productioncounter", "sensor/humidity"
};

            byte[] qosLevels = { MqttMsgBase.QOS_LEVEL_AT_MOST_ONCE,
MqttMsgBase.QOS_LEVEL_EXACTLY_ONCE };
            client.Subscribe(topic, qosLevels);

            Console.WriteLine("Subscribing on topic : machine/productionsignal1");

            Console.ReadLine();

        }

        private static void Client_MqttMsgPublished(object sender,
MqttMsgPublishedEventArgs e)
        {
            Console.WriteLine("Published success!");
        }

        private static void Client_MqttMsgPublishReceived(object sender,
uPLibrary.Networking.M2Mqtt.Messages.MqttMsgPublishEventArgs e)
        {
            string t = Encoding.UTF8.GetString(e.Message);
            Console.WriteLine("Recieved MQTT Message: " + t);

            /*    string[] parts = t.Split('&');

                    string command = string.Format(" insert into
dbo.ProductionValues([Value], [Timestamp]) values ({0}, '{1}']", parts[0], parts[1]);

                    Console.WriteLine(command);
            */
        }
    }
}
```

```

        using (SqlConnection con = new SqlConnection("Data
Source=axlier;Initial Catalog=IoTCore_Test;Integrated Security=False;User
ID=sa;Password=pwd;Connect Timeout=300"))
        {
            using (SqlCommand cmd = new SqlCommand(command, con))
            {
                con.Open();
                cmd.ExecuteNonQuery();
            }
        }
    }
}

```

## AMQP Subscriber

```

using System;
using System.Collections.Generic;
using System.Linq;
using System.Text;
using System.Threading.Tasks;

using Microsoft.ServiceBus.Messaging;
using System.Threading;

namespace AxIIot.Amqp.Subscriber
{
    class Program
    {
        static string connectionString = "...";
        static string iotHubD2cEndpoint = "messages/events";
        static EventHubClient eventHubClient;

        private static async Task ReceiveMessagesFromDeviceAsync(string partition,
Cancellation token ct)
        {
            var eventHubReceiver =
eventHubClient.GetDefaultConsumerGroup().CreateReceiver(partition, DateTime.UtcNow);
            while (true)
            {
                if (ct.IsCancellationRequested) break;
               EventData eventData = await eventHubReceiver.ReceiveAsync();
                if (eventData == null) continue;

                string data = Encoding.UTF8.GetString(eventData.GetBytes());
                Console.WriteLine("Message received. Partition: {0} Data: '{1}'",
partition, data);
            }
        }

        static void Main(string[] args)
        {
            Console.WriteLine("Receive messages. Ctrl-C to exit.\n");
            eventHubClient =
EventHubClient.CreateFromConnectionString(connectionString, iotHubD2cEndpoint);

            var d2cPartitions = eventHubClient.GetRuntimeInformation().PartitionIds;

```

```

CancellationTokensource cts = new CancellationTokensource();

System.Console.CancelKeyPress += (s, e) =>
{
    e.Cancel = true;
    cts.Cancel();
    Console.WriteLine("Exiting...");
};

var tasks = new List<Task>();
foreach (string partition in d2cPartitions)
{
    tasks.Add(ReceiveMessagesFromDeviceAsync(partition, cts.Token));
}
Task.WaitAll(tasks.ToArray());
}
}
}

```

## AMQP Publisher method

```

private static async void SendDeviceToCloudMessagesAsync()
{
    Random rand = new Random();
    int counter = 0;

    string prev = "0";
    while (true)
    {
        var telemetryDataPoint = new
        {
            deviceId = "emain2",
            previousPublishTime = prev,
            timestamp = DateTime.Now.ToString(),
            counter = counter++
        };
        var messageString = JsonConvert.SerializeObject(telemetryDataPoint);
        var message = new Message(Encoding.ASCII.GetBytes(messageString));

        Stopwatch stopwatch = new Stopwatch();
        stopwatch.Start();

        await deviceClient.SendEventAsync(message);
        stopwatch.Stop();

        prev = stopwatch.Elapsed.ToString();

        Task.Delay(2000).Wait();
    }
}

```

} }