# TCP-Based Communication Protocols as Enabling Technologies for IIoT

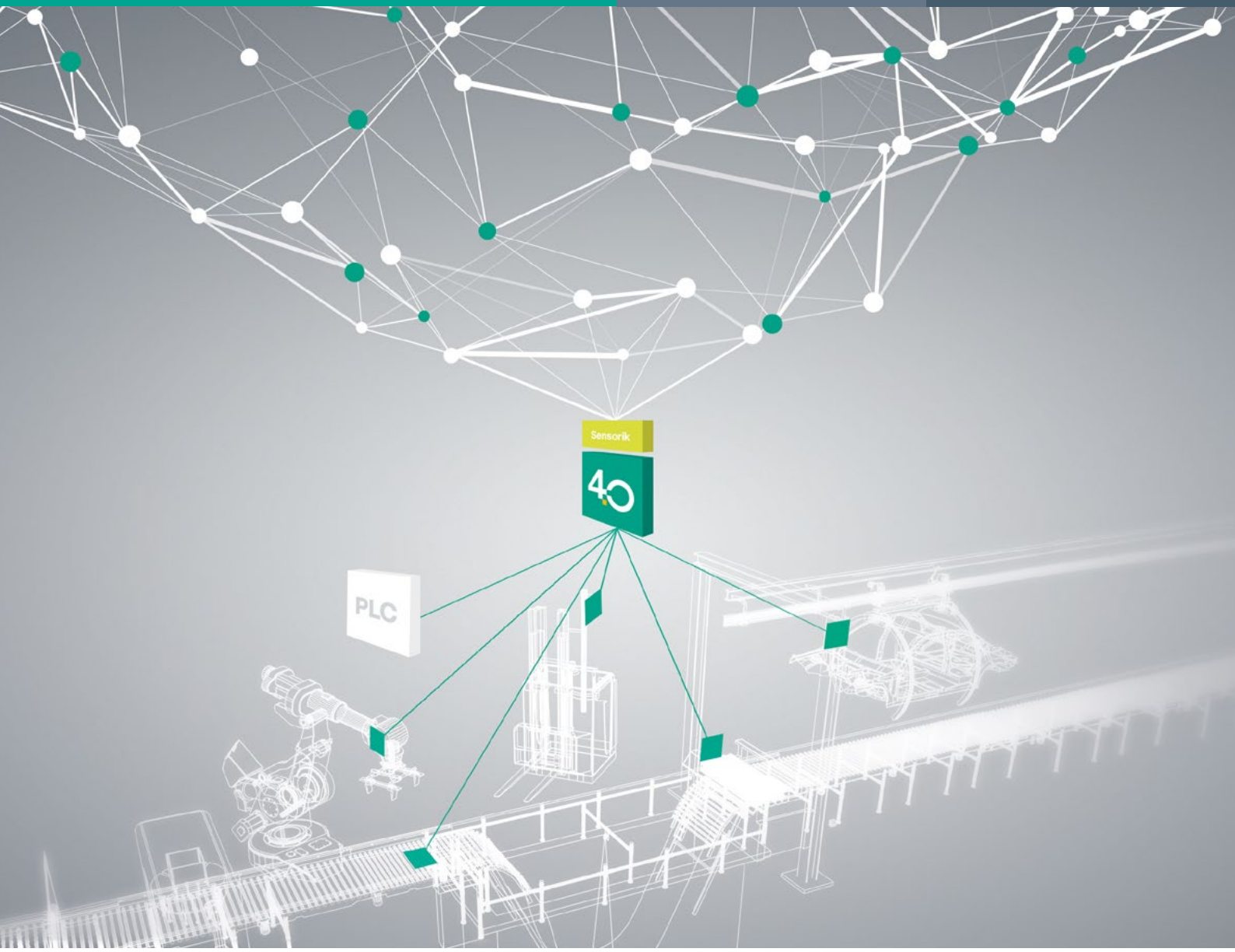
White Paper

AMQP
Advanced Message Queuing Protocol

MQTT

OPC UA

{REST:API}




Your automation, our passion.

PEPPERL+FUCHS

## The Author: Elena Petrevska

Elena Petrevska is an industrial communication engineer at Pepperl+Fuchs SE. She graduated in Automation Technologies—Electrical Engineering from the University of Applied Sciences Mannheim and in the seven years since then has worked in the industrial sector. Apart from industrial communication, Elena Petrevska has acquired knowledge in other technologies such as web development, the Internet of Things, and the cloud and is interested in and constantly learning about new technologies as they come to market.



# Contents


**Introduction** 3

**Terminology** 4

**RESTful Interface** 8

**MQTT** 10

**AMQP** 13

**OPC UA** 14

**Deep Dive Comparison** 16
OSI Model 16
Interoperability 18
Real-Time Behavior 19
Security 19
Implementation 20
Use Cases 21
Membership 22

**Conclusion** 22

**References** 23

# Introduction

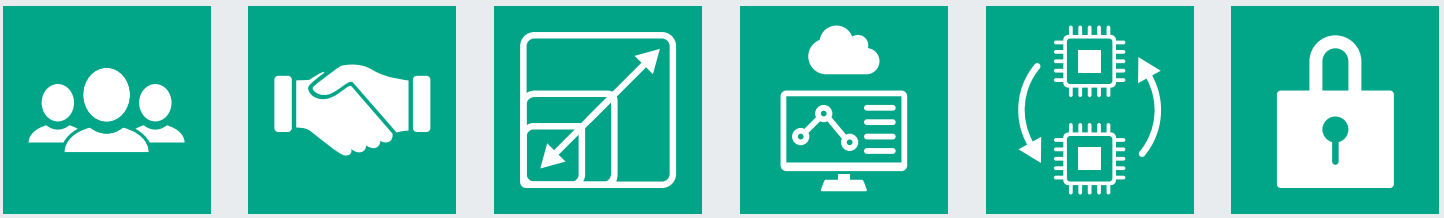
All around us, we hear buzzwords like Industry 4.0, IIoT, smart factories, and smart devices. Everyone is striving to be part of the new IIoT wave which gained momentum in recent years and is on its way to becoming the new reality. Fortunately, we are at the point in time where we have moved from buzzword bingo to the developers' offices to implement solutions.

The core idea of this new technological era is connectivity. Each device is a node in a network of many nodes. They all know and interact with each other. There is no automation pyramid, no in-between layers, but a giant mesh of nodes all aggregating with each other, working toward a particular goal. Nevertheless, having an idea and implementing it are two sides of a coin. They cannot exist without one another for the system to be complete, yet each one has its own domain.

For that idea to become a reality, all devices taking part in that network of nodes should have IIoT capabilities. There are no smart devices without smart architecture and smart communication. All devices should be armed with capabilities for communicating in an IIoT way. That means that they should communicate in a vendor-independent, standardized, scalable, IT- and OT-ready, interoperable, and secure way.

Multiple communication protocols are battling to become the new IIoT champion. Each one is promising to be the standard, the chosen one. Yet the question is, will there ever be one? Humans have moved to an era where everyone gets a medal, even for participating. Shouldn't it be the same for communication protocols? Will it ever be possible to find that champion?

In this paper, we will thoroughly focus on and compare the four IIoT communication protocols that have found their way up and are considered the enablers of IIoT and Industry 4.0. They are MQTT, AMQP, OPC UA, and the RESTful interface.



Vendor independent | Standardized | Scalable | IT and OT ready | Interoperable | Secure

# Terminology

### Client/Server

The client/server model is an architectural model for separating processes between two sides: the requester and the data provider. The client initiates the communication and requests data. The server accepts the invitation to communicate and provides data. In a typical scenario, the server is a database with lots of data and resources, and the client is an application that wants to gain access to that data. Both the client and server can create multiple parallel connections. The client can receive data from multiple servers, and the server can provide data to multiple clients simultaneously.

### Request/Response

The request/response pattern is widespread in client/server architectures. Its functioning is straightforward: the client sends a request, and the server answers with a response. It is typically implemented synchronously, which means that the client waits until the server answers the request.

### Publish/Subscribe

The publish/subscribe pattern (often called pub/sub) also applies the client/server architecture but functions differently. It consists of publishers—or providers of data, subscribers—or listeners on particular topics for data, and a broker—an intelligent queue system that transports the publishers' data to the subscribers. The difference between pub/sub and the request/response pattern is that the communication is asynchronous and that the publishers and subscribers do not communicate directly. Instead, they are only connected to the broker and do not even need to be aware of each other.

From a technological perspective, the publishers and subscribers are clients, and the broker is a server. However, unlike the standard server from the request/response pattern that waits for the client to ask, this server can also publish data on its own. Both the client and server can initiate a message exchange, making the communication between the publisher/subscriber and the broker bidirectional. There are multiple ways of how this server push function is implemented. Some examples are long polling, web push, and push notification.

From an implementation perspective, the complexity of the system lies on the side of the broker. It is the component that holds connections to all publishers and subscribers, manages lists of all subscribers and the topics they subscribed to, receives the publishers' data, and sends it to the particular subscribers. In case the subscribers are disconnected and wish to be informed of all new data on a specific topic, the broker also saves the data until they are connected again.

## TCP/IP Model

The TCP/IP model is an abstract model which describes the architectural layers in a computer network. Unlike the OSI model, it consists of four layers—link, Internet, transport, and application layer (Figure 1). Each layer has its separate task, with the ultimate goal of enabling communication from one device to another in that network.

The link layer focuses on transmitting packets through a physical medium to the receiving side. Here belong standards like Ethernet (IEEE 802.3) and wireless (personal area network, local area network, 2G, 3G, LTE, etc.). The Internet layer is responsible for preparing these packets for sending through the link layer to reach the destination no matter which route they take. At this layer, the most well-known component is the Internet Protocol (IP). The transport layer is responsible for splitting the data from the application layer in packets perceivable for the link layer. It also controls the accuracy of the packages through flow and error control mechanisms. This layer has two standards: Transmission Control Protocol (TCP) and User Datagram Protocol (UDP). The difference between the two standards is that TCP is a connection-oriented protocol and UDP is not. TCP, being connection-oriented, requires a handshake between the two sides and provides data control mechanisms to acknowledge and check the data's validity. It is heavyweight but secure. UDP, on the other hand, is very lightweight, does not require a connection between the two sides, and works on the principle of "fire-and-forget"—it fires the data and forgets it, without acknowledging that the data is valid. It also performs validity checks but ignores erroneous packets.

As discussed in [1], TCP was ignored as a standard enabling the IoT revolution because it seemed that TCP could not stand up to the requirements. How can tens of billions of devices (sensors, actuators) be connected in one mesh, all of them in need of a handshake, and at the same time with all of the constraints (low-power, error-prone connections, memory) they bring with them? It seemed unrealistic to use TCP. UDP won the game. Or at least, it seemed like it. Implementations started in the UDP world—Constrained Application Protocol (CoAP) and IPv6 over Low-Power Wireless Personal Area Networks (6LoWPAN) emerged. However, soon enough, CoAP encountered the first challenges connecting to enterprise networks and networks in geographic regions with otherwise challenged connectivity, and it was decided to develop CoAP over TCP [2]. That was a game changer. At the same time, the TCP standards were improved and optimized for the IoT environments and fought their way up to the throne. They are now the leaders of IoT and IIoT/Industry 4.0 and the focal point of observation in this paper.

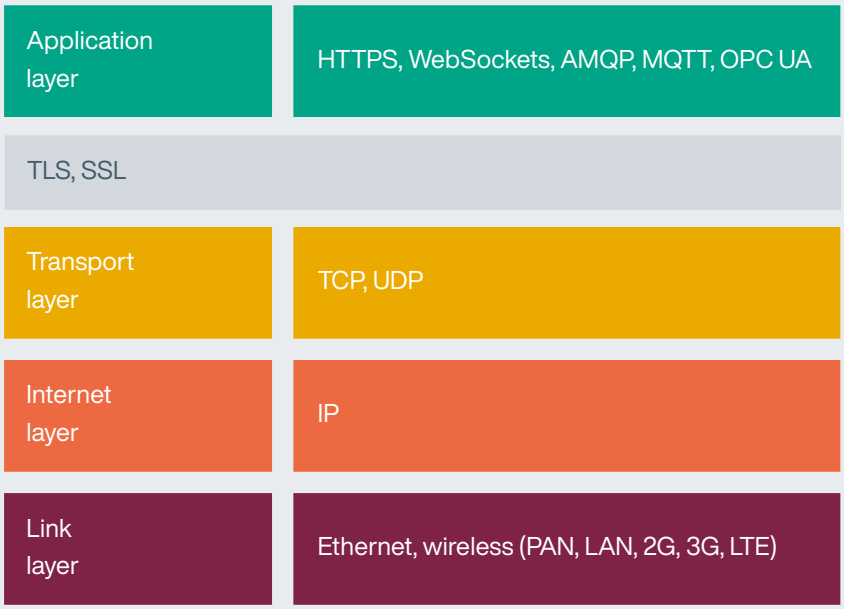



Figure 1: TCP/IP model

Continuing with the TCP/IP model, a small layer is visible between the transport and application layers. Yet its thinness does not measure up to its importance. It is a security layer that is not part of the TCP/IP model (no surprise since it was developed in the 1960s) and provides security over that computer network. It consists of the Transport Layer Security (TLS) and its predecessor Secure Sockets Layer (SSL).

On the top lies the application layer. It is the layer where many different tasks occur, such as providing information like login data, acquiring process data, encoding the data, and even encrypting it. This layer is the layer where the technologies like HTTPS, AMQP, MQTT, and OPC UA have their core components and will be elaborated on in this paper (Figure 2). For a more straightforward explanation of the complexity found on the application layer, it is subdivided into four layers—transport, security, data encoding, and application. On each of these new layers, MQTT, AMQP, OPC UA, and the RESTful interface define new components and add existing ones like HTTPS, WebSockets, and security mechanisms as enabling blocks for their functionalities.




Figure 2: TCP communication protocols are built on top of the application layer

### Challenge–Response Authentication

The challenge–response authentication is a family of authentication mechanisms in which, while creating a connection between two devices, one device sends a "challenge" to the other device. That receiving device must respond with a valid answer to be authenticated [3]. The challenge can be in the form of multiple algorithms: Challenge–Handshake Authentication Protocol (CHAP), OATH Challenge–Response Algorithm (OCRA), Salted Challenge Response Authentication Mechanism (SCRAM), etc. This authentication method is relevant for MQTT.

### SASL

Simple Authentication and Security Layer (SASL) is a method for authentication in connection-based protocols [4]. It also includes various authentication mechanisms, and the devices can choose which ones they want to implement. At the connection initialization, the two devices inform one another which SASL mechanism they support using a command and, after the authentication is successful, start communicating. Some of the SASL authentication algorithms are ANONYMOUS (unauthenticated access), PLAIN (clear-text password), CRAM-MD5 (challenge–response scheme), OAUTHBEARER (OAuth bearer tokens), SCRAM, etc. This authentication method is relevant for AMQP.

# RESTful Interface

Representational state transfer (REST) is a set of architectural constraints that simplify and harmonize the communication between devices in a network. Roy Fielding developed it in his PhD dissertation in the year 2000 [5]. REST can be implemented on top of any protocol but has found its way into the Internet networks and is best known for setting architectural constraints on the HTTP(S) specification. HTTP is an extensive request/response communication protocol on the application layer. However, if everyone used HTTP as they wanted, there wouldn't have been any synchronization, and the components wouldn't have been able to talk to each other. REST came to the rescue and harmonized HTTP usage.

The principle behind REST, as the name suggests, is transferring the (server's) state in a representational manner (to the client). There are two important points worth mentioning here. First, the server is stateless. It does not save state information about if and when it was connected to a client. It just sends the momentary state to the client, and the client has the freedom to create a stateful application based on the separate states it receives from the server. Second, the server can send the resources in multiple representational formats to the client. Every data unit that the client can access in REST is called a resource and can be reached through a unique access point called a Uniform Resource Identifier (URI). Every resource has its own URI. The appeal of REST is that the resources are decoupled from their representation. The server can store the resources in any format (in a database, CSV file, or an object-oriented class). It can still deliver the resources in one or multiple representational formats like binary, JSON, XML, YAML, or JPEG. That means the client accesses the representation but not the resource itself.

Any application or interface that uses REST is called RESTful. For an interface or application to be RESTful, it should follow these five constraints:

- **Client/Server:**
  The communication should take place between a client and a server. As previously mentioned, the client is the seeker, and the server is the provider of information. For example, in Figure 3: The client asks the server for the temperature, and the server responds with a message containing the temperature in JSON format. It is essential to mention two crucial components that the client's request should contain—a URI and HTTP method. The HTTP methods are actions that are requested from the client to the server. There are six standard HTTP methods (GET, PUT, POST, DELETE, HEAD, and OPTIONS), and each one is used in a specific situation. As examples, GET should be used to receive the state of a resource, PUT to update a resource, POST to create one, and DELETE to delete a resource.

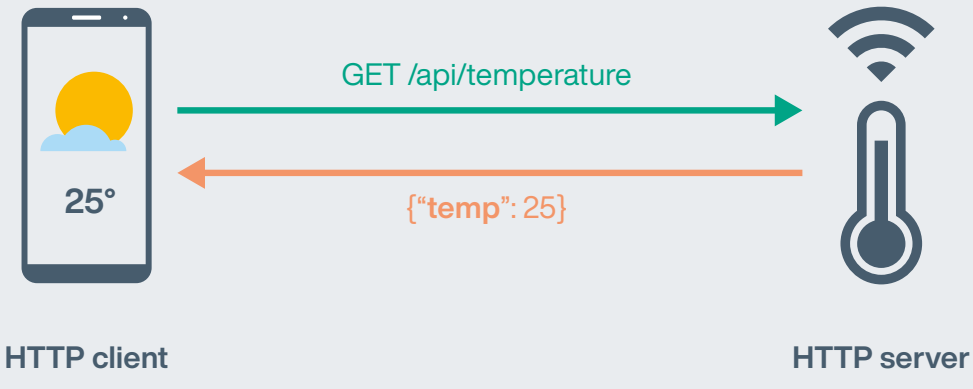



Figure 3: Typical RESTful application

- **Statelessness:**
  The server does not remember with which client it talked. That shifts the responsibility of saving the states to the client side and makes the server lightweight.

- **Cacheable:**
  The server contains information if its resources are cacheable, and if they are, the resources include version numbers so that the client receives information about their validity and history.

- **Layered system:**
  No matter how many layers are between the client and the server, no matter if there are proxies between them, they should be able to communicate with each other. There may be multiple layers like the security layer, caching layer, and load-balancing layer, but it should not immobilize the client's and the server's message transfer.

- **Uniform interface:**
  There are multiple constraints interconnected in this one regarding the interface universality:

  - Each resource should have its unique ID in the form of a URI.
  - Once the client has a representation of a resource and its metadata (and a permission to do so), it can add, delete, or modify a resource.
  - The exchanged messages should be self-descriptive and provide enough information on how to process the resources.
  - Resources are linked together, and the client can get information on the URIs of all resources by accessing a specific URI.

# MQTT

Message Queuing Telemetry Transport (MQTT) is a lightweight, binary, simple, open, and easy-to-implement publish/subscribe messaging protocol. It is a certified OASIS and ISO standard (ISO/IEC 20922:2016), and the OASIS technical committee manages it. MQTT's first appearance was in 1999, as a technical solution for the problem of monitoring oil pipelines in a bandwidth- and battery-power-efficient way. Fourteen years later, in 2013, IBM submitted MQTT to the OASIS specification body and it was released as an OASIS standard in 2014 [6]. Since then, it has conquered the industrial market, especially in areas where constraints like small code footprint, minimal overhead, and limited network traffic are required.

The basic functionalities for MQTT are as follows (Figure 4): The MQTT clients (both publisher and subscribers) connect to the MQTT broker. Some MQTT clients subscribe to a particular topic, in this case to the temperature value. The broker remembers that, and once the publishing MQTT client delivers some data to the broker, the broker transmits that message to all the subscribed MQTT clients. The considerable benefit of this type of topology is in the decoupling of the publishers and subscribers. Instead of many connections between them, the broker is responsible for transporting the data from one end to another.

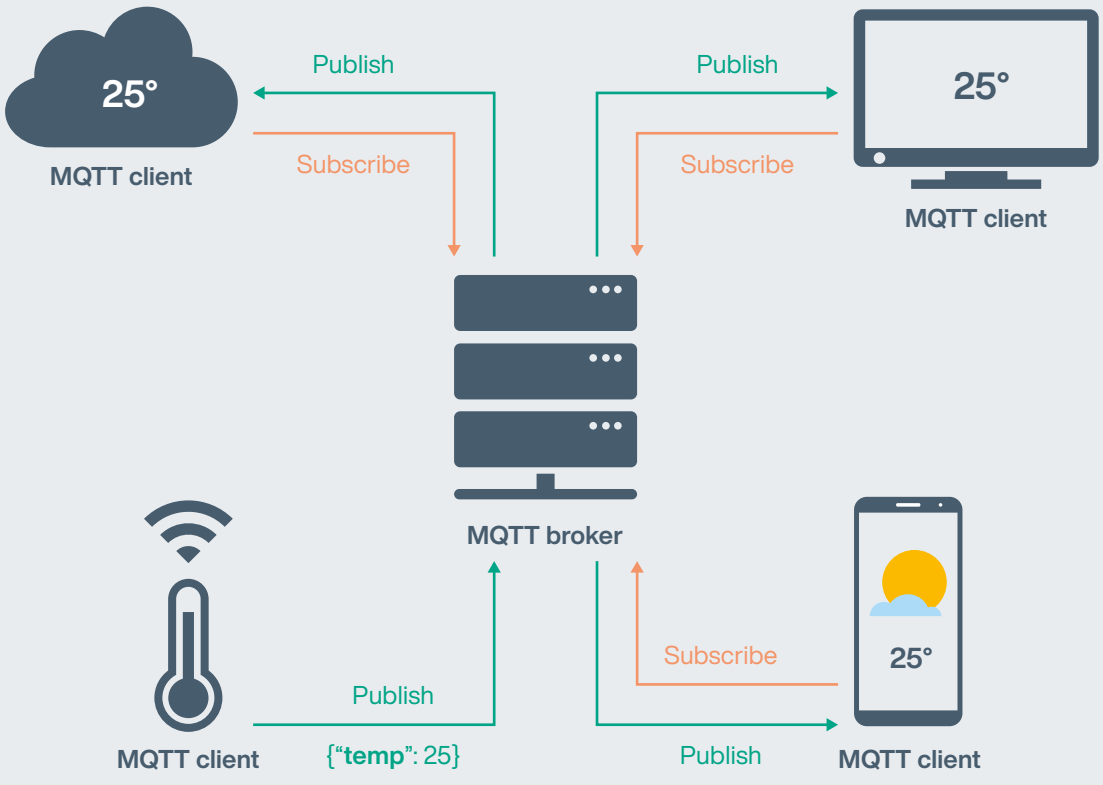



Figure 4: Typical MQTT application

**In the MQTT specification, there are multiple features worth mentioning:**

- **Topic:**
  A foundational block in the MQTT protocol. Each message sent through the wire consists (apart from header information) of a topic and a payload. The topic is the information with which the broker can filter through messages and to which subscribers subscribe. Topics can consist of multiple levels; for example, "home/sensor1/identification" consists of three topic levels and "home/sensor1/data/temperature" consists of four topic levels. Each topic level is separated by a forward slash and the topic structure is case-sensitive. Topics can also include wildcards on individual and multiple levels. Using a single-level wildcard is implemented by using the plus symbol and could look like this: "home/+/data/temperature." What this does is collect all temperature values from all sensors in the house, and publish them on subscribed MQTT clients. The second type of wildcard is the multilevel one, and it uses the hashtag symbol. An example would look like this: "home/sensor1/#," as a result of which all data published from sensor1 will be transmitted to the subscribed MQTT clients.

- **Quality of Service:**
  This is a feature for defining the delivery assurance for each message. The idea behind it is to define different levels of security that the message will arrive at the subscriber side. Those levels are: QoS 0 (at most once), QoS 1 (at least once), and QoS 2 (exactly once). QoS 2 is the most advanced quality level and requires the most amount of communication acknowledgments and implementation effort.

- **Session awareness:**
  There are multiple mechanisms in MQTT for making sure the connection between the broker and publisher/subscriber is intact. The most elemental mechanism is the keep alive message. The client sends a keep alive message from time to time to assure the broker that the connection is still up. This helps the broker detect if the client has gone offline. In addition, during the initial connection process, the client provides a last will and testament (LWT) statement to the broker. This message is sent from the broker to the remaining network clients after the client has gone offline, so that they are informed of that client's absence.

- **Persistent session:**
  Where a persistent session is established between the broker and a subscriber, the broker will store the messages until the subscriber is ready to receive them. This feature is especially useful for unstable networks.

In March 2019, **MQTT 5** was released and was also OASIS and ISO approved [7]. It consisted of multiple new features and changes on the previous version, some of which were, unfortunately, breaking changes. The new features of MQTT 5 include:

- **Shared subscriptions:**
  Multiple clients can share the same subscriptions. This feature is helpful for better load balancing and work distribution.

- **Additional status codes:**
  More return codes were specified, which results in better error reporting and finding the reason for failure.

- **User-defined key-value properties in the header:**
  Possibility for expanding the MQTT header with an arbitrary number of key-value pairs, helpful in carrying metadata.

- **Additional control package for security:**
  Authentication was possible until MQTT 3 through username and password in the CONNECT control package. In MQTT 5, apart from entering the password in the CONNECT control package for token-like security, there is another control package called AUTH for nontrivial authentication mechanisms.

- **Bidirectional DISCONNECT:**
  In MQTT 3, only the client could send a disconnect request to the broker. The broker was not able to disconnect itself from the client. Starting with MQTT 5, the broker can also send a disconnect request to the client.

- **Topic aliases:**
  It is now possible to create topic shortcuts or aliases in the form of a number to shorten the message length. This is especially useful for scenarios where smaller packages are required, like simple sensors and actuators with a small memory footprint.

- **Message expiry date:**
  The client can set a message expiry date so that it warns the subscribers that the message is not valid after that date.

# AMQP

The Advanced Message Queuing Protocol (AMQP) is an interoperable asynchronous publish/subscribe messaging protocol. It was first developed in 2003 by the bank JPMorgan Chase, which grew into a working group in 2005, including companies like RedHat, Microsoft, Software AG, Cisco, etc. There were multiple experimental releases, 0.8, 0.9.1, 0.10, and AMQP finally got its 1.0 version in 2011, when it was also released as an OASIS standard [8]. Since then, it has been developed under the OASIS umbrella. It was also ISO/IEC certified in 2014 (19464:2014). AMQP has been primarily used in the enterprise and business sectors, but it has the potential to become part of the industrial sectors as well. It is very often compared with MQTT. On the surface level, they look similar to each other, but underneath the surface, AMQP, unlike MQTT, defines a complete ecosystem (more on this in the later sections).

In the AMQP topology, both the clients and the brokers are called containers (Figure 5). Containers are entities that constitute multiple nodes. Each node can be a producer, consumer, or broker. The producers generate, the consumers receive, and the brokers store and forward messages. That means that one client can contain both consumer and producer elements. Inside an application, these nodes can be either flat or hierarchically connected.

On the outside, nodes are connected through links. A link is a unidirectional route between two nodes formed on top of a session. Each link has its unique identifier, and if the connection between the containers is interrupted, the links can be recovered after reconnecting.

Another exciting aspect of AMQP is that apart from having brokers on the network, it can also have routers. The routers only implement the AMQP transport layer and carry out the transmission tasks, without queuing the messages. That means that the router creates a synchronous connection between the producer and the consumer. The consumer and the producer then communicate with each other in a standard request/response way. This feature is useful in cases where the producer and consumer are geographically separated from one another. In an AMQP topology, the brokers and the routers can exist in parallel and do not have to (but can) be integrated into one device.

Regarding the message transmission, AMQP has two delivery mechanisms: settled (at most once) and unsettled (at least once). The transmission format for the whole (annotated) message is XML, but the message body can take on various encoding formats like binary, JSON, Avro, or XML.

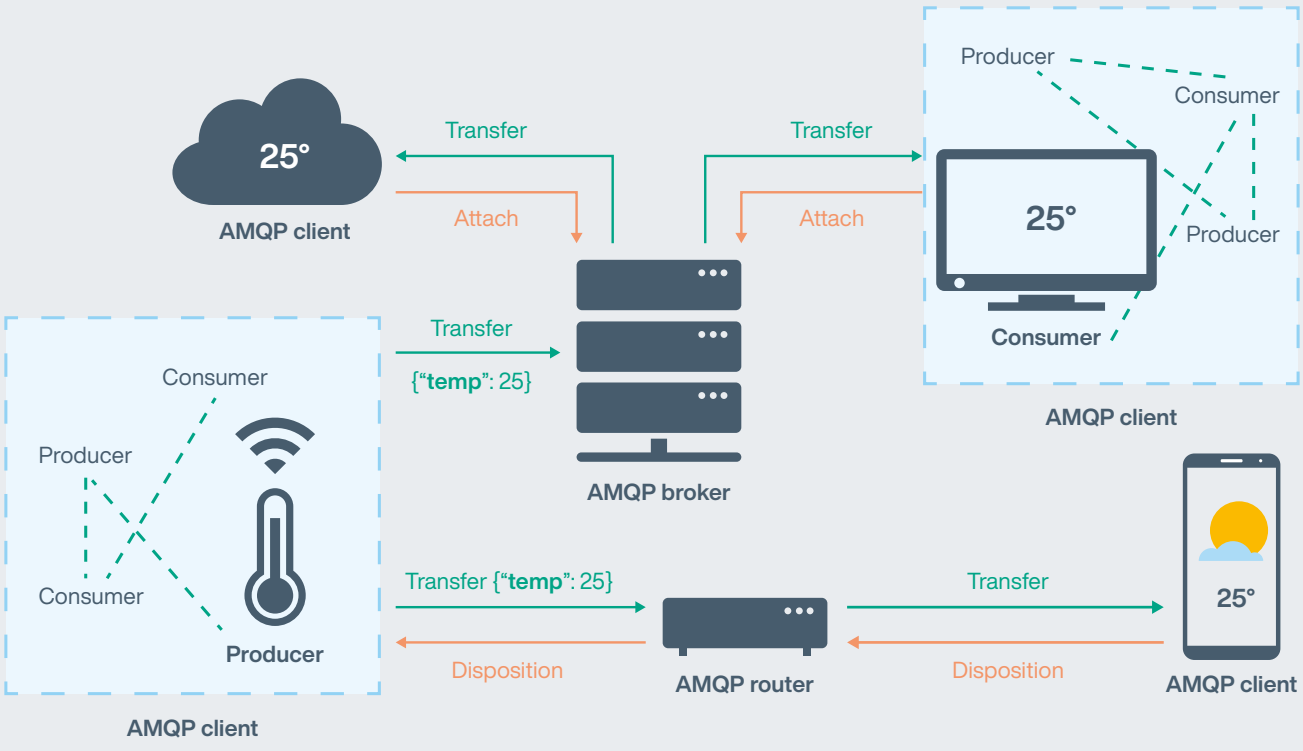



Figure 5: Typical AMQP application

# OPC UA

OPC Unified Architecture (OPC UA) is a platform-independent, service-oriented architecture. As the successor to Classic OPC, OPC UA's task was to correct the shortcomings of Classic OPC, among which the most important were the dependencies on Microsoft and COM/DCOM. The same organization which developed Classic OPC—the OPC Foundation—is also responsible for developing and managing OPC UA. The first specification release was in 2008 and was IEC certified (IEC 62541) [9].

OPC UA, being named an architecture and not a communication protocol, provides a complete ecosystem for implementations. And that ecosystem is so flexibly built that if new technologies are developed tomorrow, OPC UA will be ready to bind and leverage them to the existing system. For example, in 2018, OPC UA pub/sub was released (OPC UA was primarily request/response driven until then). Additionally, the Time-Sensitive Networking (TSN) standards that allow deterministic timing behavior are the priority at the moment. This motivation to always stay on track is very much supported by the OPC Foundation's constant engagement in multiple technical and marketing activities.

An essential aspect of OPC UA is that there are no device description files. Each device brings all information with itself and does not depend on external files for communication, identification, or data accessibility.

Yet that is not even the most fascinating part about it. The most significant benefits OPC UA brings to the industry are its data modeling, information models, and various consortium activities like technical working groups striving toward the harmonization and communication of all devices in an entirely industry-universal way.

The foundational elements of the OPC UA data modeling are the nodes and references. Each data unit is called a node, and all nodes connect through references. The nodes and the references can be of various types. For example, a node can have the type Object, Variable, Method, ObjectType, DataType, etc. The references have types too, but are grouped on a higher level into hierarchical and nonhierarchical groups. The nodes and references are collected in an address space, and all nodes are accessible through nodeIDs. The nodeID is like an address on which that node (data information) exists.

The information models are building blocks for the OPC UA application. They provide standard node definitions which promise better harmonization among devices. There are multiple levels of information models. There are foundational models defined by the OPC Foundation like Data Access (DA), Historical Data Access (HDA), Alarms and Conditions, and Programs, which provide knowledge of which features an OPC UA application supports. DA is the standard process and data access parameter. HDA adds features for storing multiple measurements from the same node for statistics. Alarms and Conditions provides the possibility for sending events in the event of emergency notifications, and Programs adds the option to define simple state machines. Due to OPC UA's scalability, it is possible to specify only the Data Access information model, which is especially useful for devices with limited memory and processing power. On top of these four information models, the OPC Foundation created one more model, called the Device Integration (DI) information model, which is the foundation for many additional information models from technical working groups like FDI, PLCopen, AutoID, OPC UA for IO-Link, AutomationML, etc. These information models are helping the harmonization of devices enormously because they implement the same information on the same access points (nodeIDs). That means replacing one device with another, even from another vendor, is very simple.

## OPC UA Pub/Sub

The OPC UA Pub/Sub extension has its own message format (NetworkMessage) and uses different communication protocols than the client/server implementation. Specifically, it uses MQTT, AMQP, OPC UA UDP, and OPC UA Ethernet. The intriguing part is that these four protocols are situated on three different TCP/IP model layers—the link, transport, and application layers.

Figure 6 shows the two patterns—request/response and pub/sub are presented next to one another. The OPC UA application contains both an OPC UA server and an MQTT client and communicates with two parallel TCP connections. It either publishes data to the MQTT broker or receives a request from the OPC UA client. Nothing stops the application from communicating with them in parallel; only two TCP connections are necessary. The advantage is that the data is saved only once but is transmitted through two separate channels.

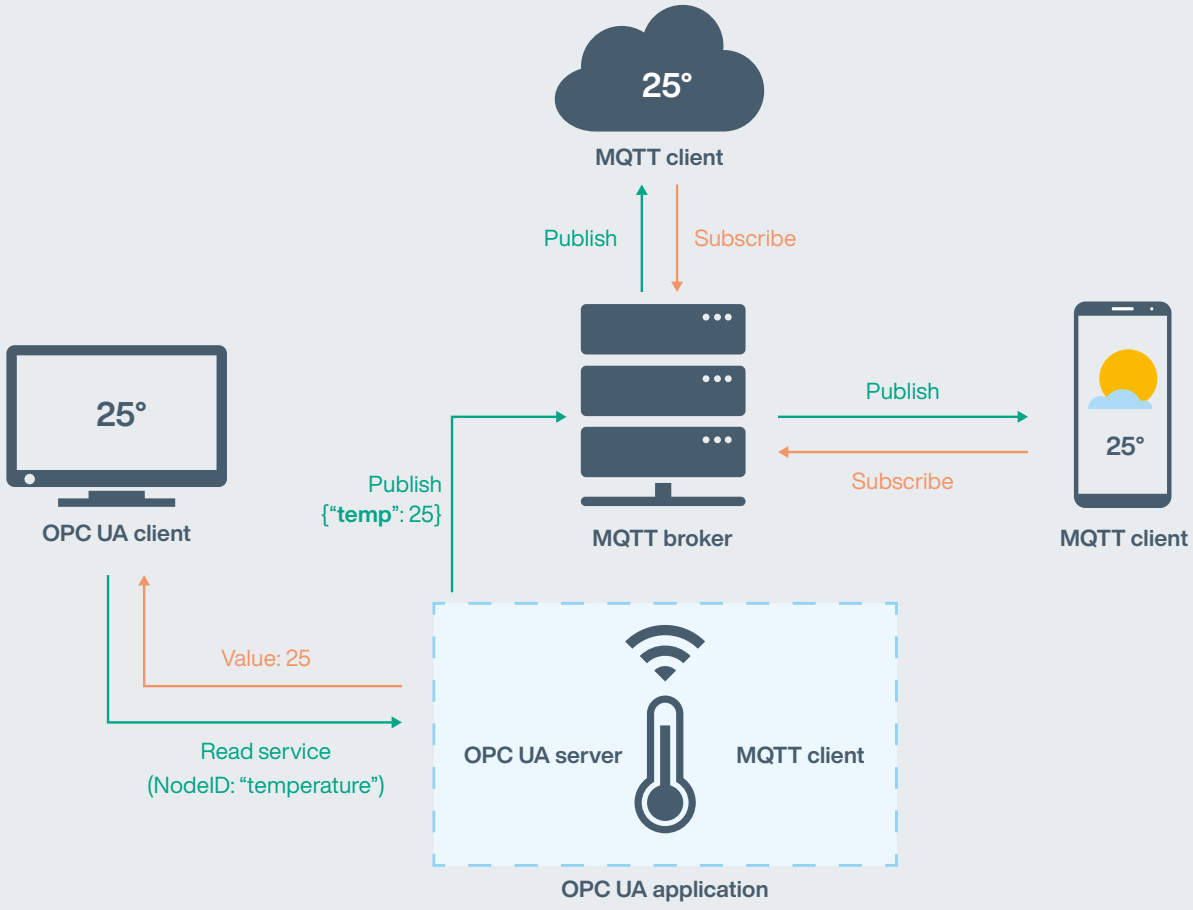



Figure 6: Typical OPC UA application

# OSI Model

It is worth noting once more that the RESTful interface, MQTT, AMQP, and OPC UA are all built on top of TCP as a transport layer, and are standardizing only the application layer.[1]

As shown in Figure 7, MQTT, AMQP, and the RESTful interface differ in the transport mechanisms and share the same data encoding features. All of them can transmit data formats like binary, XML, JSON, etc. Additionally, MQTT and AMQP are similar to each other on the transport level in that they both require a bidirectional connection where the server can push messages. The only difference is that AMQP defines its network protocols, while MQTT relies on external transfer protocols like WebSockets. On the one hand, it is advantageous to have a self-defined network protocol mechanism as is the case for AMQP, because the specification is a complete system including all necessary components for the system to function. On the other hand, being flexible like MQTT is advantageous if there is a well-established and widespread transport-level protocol that is already integrated into many components, as is the case for WebSockets. In that case, it will be simpler to integrate MQTT into the existing application than to implement the complete AMQP specification. The RESTful interface on the other hand is implemented on top of the HTTPS protocol. Unlike MQTT and AMQP, it uses the request/response pattern and requires a unidirectional connection between the server and the client.

[1] The only exception is OPC UA Pub/Sub with Ethernet II and UDP, but this will be ignored for this research.

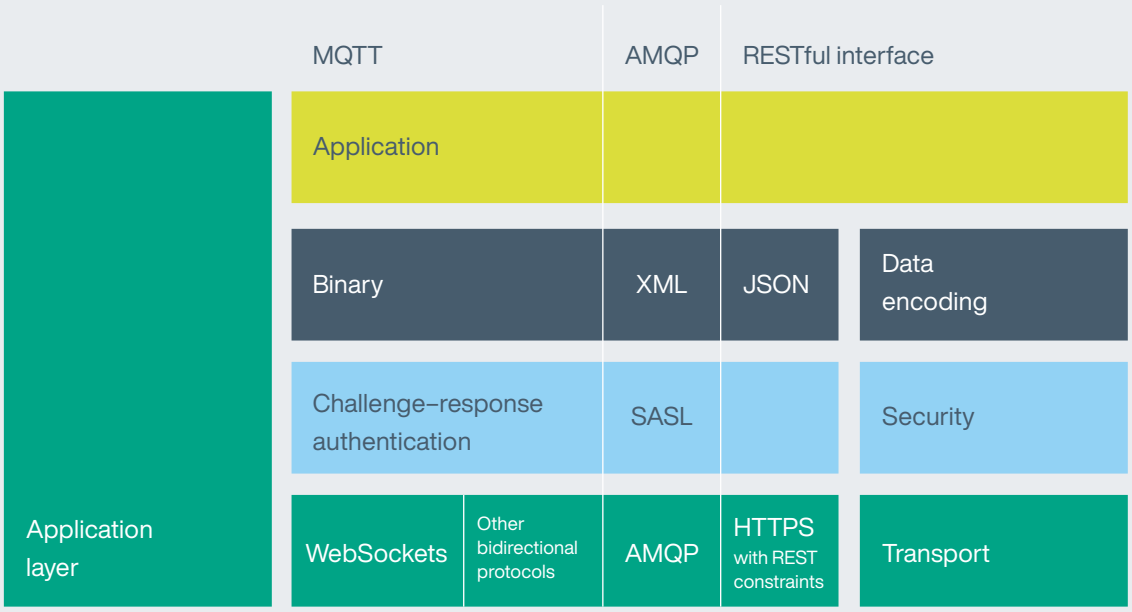



Figure 7: Deep dive in the application layer: MQTT, AMQP, and the RESTful interface

OPC UA is represented alone in Figure 8 for three reasons: it adds its own security features, defines additional data encoding formats, and uses AMQP and MQTT as transport level protocols. The most common OPC UA request/response application implementation is based on the TCP binary path—which includes the raw transport level, UA Secure Conversation, and binary messages. This is the required implementation in a request/response pattern. Additionally to the request/response pattern, there is a web path, which however is not very common because of its time limitations. For the newer pub/sub functionalities, OPC UA uses AMQP and MQTT on the transport level and defines its own JSON and UADP message formats. UADP uses the optimized UA binary encoding and provides security for pub/sub. It defines optional header fields and different message types as well as data encodings. The JSON format allows the exchange of data comprehendible for the web and other enterprise software that uses this format.

However, whether or not it makes sense to use these subsets of MQTT or AMQP through OPC UA depends on the use case. If an OPC UA server or client already exists in the infrastructure, and there is a need to connect through MQTT or AMQP to the "other side," then yes, it does make sense. If the scenario is to transfer data only through MQTT or AMQP, there is no need to build the OPC UA application supporting MQTT or AMQP.

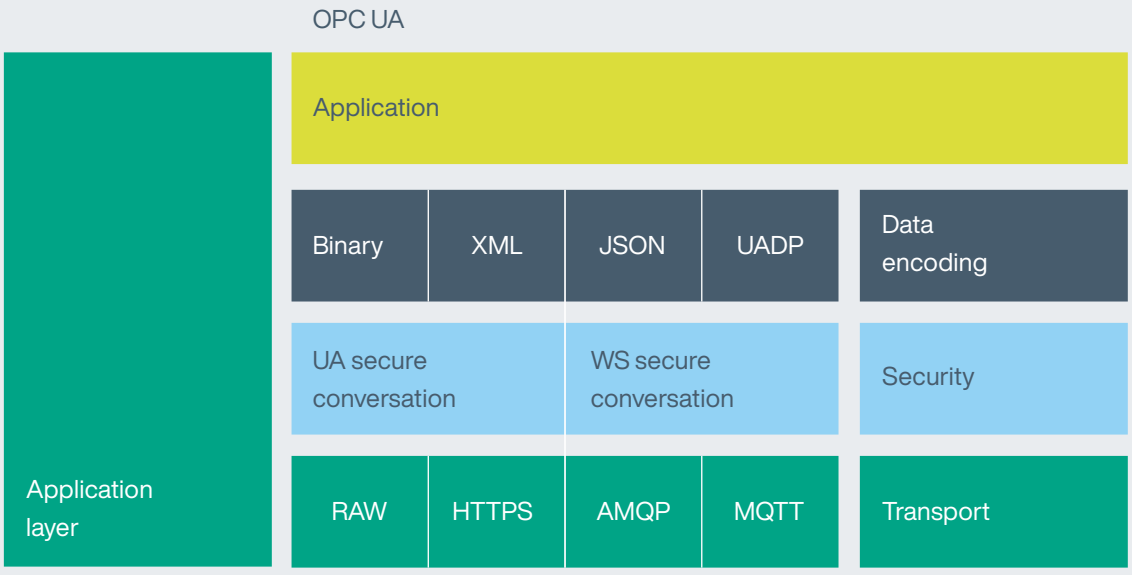



Figure 8: Deep dive in the application layer: OPC UA

Deep Dive Comparison

# Interoperability

When talking about interoperability, two types come to mind: machine-to-machine and human-to-machine. Machine-to-machine interoperability focuses on the ability of the devices to communicate with each other without any human interaction with the system. Human-to-machine interoperability is the ability for humans to understand the functionality of machines and interact with them with the least possible effort and vice versa.

A very important enabler of interoperability is the standardization on the data level. It is one thing to standardize a protocol on a transport level, another thing to standardize a protocol syntactically, and completely another thing to standardize a protocol semantically. Standardizing a protocol syntactically means defining the interfaces and the data types. Standardizing a protocol semantically on the other hand means defining data instances that are the same in all the applications.

MQTT is the most lightweight of all protocols and does not define any standardization levels. The payload (message) gets sent to a specific topic, and misspelling a topic or sending data in another format than desired is not considered a mistake. Since MQTT 5, metadata can be sent in the headers; however, it is possible for each device to have a different naming scheme and different metadata information. That makes MQTT insufficient for machine-to-machine interaction without a human setting all of the topics and messages beforehand. For human-to-machine interaction, the human should first read from a manual which topics are available and how their data is structured before interacting with the machine. The machine cannot say by itself which topics are available for subscriptions.

AMQP is also relatively lightweight and the only interoperability feature it defines are data types. Also, using the links, at least one can make sure that a particular producer node is available, which improves the machine-to-machine capability. Yet that sounds more like a guessing game and less like a scenario where the devices can talk freely with each other. For human-to-machine interaction, AMQP is very similar to MQTT. The human should open the manual and search for the available producer data.

The RESTful interface has multiple standardization possibilities on a syntactical level like OpenAPI, RAML, and API Blueprint. They are standards for describing the RESTful interfaces for both humans and computers and allow them to interact with each other without ever opening a manual. That makes the RESTful interface both machine-to-machine and human-to-machine capable. For machine-to-machine interaction, both machines can process the REST APIs from their communication partners and can access data without human interaction. Only in the case that a meaningful application is to be created (semantical interoperability) does a human have to interact with the system. For human-to-machine interaction, multiple representational tools make understanding and interacting with the REST API very convenient.

OPC UA has many interoperability features—syntactic and semantic. Syntactically, all devices use the same types and services to communicate. On top of that, they use information models which are all saved in the same format—a nodeset file. The nodeset file is a collection of all nodes and references that are defined on top of the basic modeling rules in OPC UA, a machine-understandable format of the data defined in an information model. The nodeset files provide syntactic as well as semantic interoperability. From a syntactic perspective, these files have the same structure and can be parsed from any OPC UA application. From a semantic perspective, many working groups work toward synchronizing their parameter and process data names, their method availability, events, etc., so that devices act and feel similar to one another. An example is the AutoID information model, which strives to unify all identification devices like RFID readers and barcode scanners. They can have additional vendor-specific customizations, but the basis should be the same. That simplifies the implementation of the OPC UA client because it does not need to have a connector for each special variant. In conclusion, OPC UA is machine-to-machine capable. With the usage of visualization tools for the information model representation, it is also human-to-machine interoperable.

Deep Dive Comparison

# Real-Time Behavior

Because of its minimal overhead, MQTT achieves fast transportation times. Theoretically, where there is no lag between the publisher and the broker, and the subscriber is connected to the broker and listens actively, the promised "near-real time" and "delay-free" delivery could be achieved.

AMQP, being also a pub/sub communication protocol, should also achieve relatively fast transportation times. It will not be as fast as MQTT because its payload is bigger, but near-real time should again be possible.

OPC UA has the possibility to compete in real time after expanding to the pub/sub world. With its request/response pattern, it would be nearly impossible to compete with MQTT and AMQP.

The RESTful interface, with HTTP being a request/response communication protocol, does not have real-time capabilities. The speed of the data actualization depends on how much the client polls the server, and with too much polling, it may congest the network.

Deep Dive Comparison

# Security

Security is a multilayer concept on the transport and application layers. For all four protocols, it is recommended to use TLS on the transport layer. On the application layer, each one has its own recommendations.

MQTT provides the possibility to authenticate with a username and password in the connection initialization phase. Since MQTT 5 it also allows the use of just the password field for a token transmission and relies on further security mechanisms like the challenge–response authentication. AMQP relies on the SASL as security standard. OPC UA follows the Web Service Security Specifications for its WS-SecureConversation and creates a binary variant from it for the TCP binary path with the name UA Secure Conversation. In addition, it provides username/password, X.509 certificate, and issues token security mechanisms, and has features for auditing messages for logging which client changed what value at what time. And finally, the RESTful interface does not have security by design, because at the time it was written security was not a priority. Still, there are many best practices available, like authentication with username and password, JWT token, and OAuth security mechanisms.

# Implementation

All four communication protocols have implementations in multiple programming languages: C, C++, JavaScript, Python, .NET, and many others. There are numerous libraries available, both in the open-source and the commercial sector. Collections of implementations for many flavors can be found in the following links (the focus lies in the open-source sector): MQTT [10], AMQP [11], OPC UA [12], and the RESTful interface [13].

Depending on the programming language and the library used, as well as the activated features, the application can take from tens of kilobytes to many megabytes. The minimum application sizes are listed in Table 1. MQTT with its lightweight characteristics uses the least memory of all protocols.

| | MQTT | AMQP | OPC UA | RESTful interface |
|---|---|---|---|---|
| **Minimum application size** | 3.6 kB (1) [14] | 100 kB (1, 2) [15] | 100 kB (1) [16] | 32 kB (1, 2) [17] |

(1) Without TLS/SSL.
(2) It was difficult to find any size information, and this one refers to one TCP connection.

Minimum application sizes for MQTT, AMQP, OPC UA, and the RESTful interface

These kinds of sizes can only be achieved with an appropriate scaling of the protocol. MQTT and AMQP are as lightweight as it gets and there is no room or need for optimization, apart from including/excluding security features. The same applies to the RESTful interface. The only protocol which can expand (because it has many features) is OPC UA. Apart from the standard security scaling, which just means turning off security (and is recommended neither for OPC UA nor for the rest of the protocols), OPC UA can be scaled in at least two ways—by choosing an appropriate OPC UA profile for small embedded devices (Nano Embedded Device Server Profile) which limits the number of TCP connections to one, as well as by limiting the application to only the basic information models like Data Access (DA).

The next thing to consider is the implementation effort. MQTT is the simplest to implement. This is especially the case when WebSockets is already available in the system and the data exchange format (topic and payload) depends only on one or two parties. The heaviness of MQTT comes on the side of the broker, but the customers and developers will spend most of their time implementing the client side, where the effort is negligible.

AMQP on the other hand is relatively complex. It offers a whole ecosystem that needs to be implemented. There are so many things that should be taken into consideration and it is recommended to make use of a library that already has at least the AMQP transport protocol implemented.

OPC UA is also complex by nature, especially when it comes to building a complete information model. It is therefore recommended to use already existing information models that fit the use case. From the protocol and security view, it is also advised to use an existing library and build on top of it.

The RESTful interface is not as simple as MQTT and not as complex as OPC UA and AMQP. An application can be developed fast, especially if HTTP is already available as a protocol. The challenge grows from an implementation perspective when the REST API includes many resources.

However, the development does not stop there. How useful is an application that is not tested and validated? How robust is it? Can it be installed in the industry? The forerunner in this category is OPC UA. It has three levels of testing: self-tests using the UA CTT tool, interoperability workshops, and independent third-party testing labs. The other three protocols rely on self-tests. The RESTful interface has REST API validators, MQTT and AMQP have “test it yourself” labs [18] [19].

# Use Cases

Before talking about the separate protocols on their own, it is very important to say that use cases should in general be grouped into request/response and pub/sub. request/response use cases include scenarios where a direct connection is necessary. It is a proven and reliable communication pattern and the functionality is simple—the client requests and the server responds. Pub/sub on the other hand is very useful when multiple, if not millions of devices have to be connected in a network and communicate with each other. Instead of opening millions of connections to each component in the network like it would be the case for request/response, they will connect to the broker. The complexity in the broker will increase, but that is solvable by distributing the work to multiple interconnected brokers.

Coming back to the four communication protocols discussed in this paper, some general use cases can be derived. For example, MQTT is perfect where a simple, lightweight solution is sufficient. That is the case when gathering data from multiple sensors, or sending the same information to many sensors from the same type simultaneously, or where low power and a very small memory footprint are required. MQTT has been implemented in various scenarios like connected cars, military use cases, industrial device meshes, controlling of drones, etc. [20].

AMQP is mostly being used in the business and enterprise sectors, but it has its strengths in lightweight implementations where a whole solution, independent of external application layer protocols like HTTPS or WebSockets, should be implemented. As an example, similar to MQTT but a bit more general, it is for monitoring and globally sharing updates, enabling offline clients to fetch data at a later time, monitoring of risk-related data, controlling internal operations, and in-house event distribution [21].

OPC UA shines where the data shared should be standardized and harmonized, where devices should be easily replaceable because they share the same information models, where events, alarms, and historical data are useful, and even where there is a mixture of request/response and pub/sub communication. No wonder OPC UA has been installed in millions of machines and factories around the world, from transportation and automotive to chemical manufacturing [22].

The RESTful interface has been a standard in the Internet sector for a long time, and most users are familiar with it. It is perfect in situations with direct communication between two parties and communication should be easily understandable and usable, even for a human. There are very few companies that do not have a RESTful interface definition either for their devices or for the services they provide. Pepperl+Fuchs also has a RESTful API for its sensors [23]. One of the assets of REST is that it can easily be parsed in the backend of a website and presented in a user-friendly way.

# Membership

Behind a successful communication protocol there is a whole organization—OPC UA has the OPC Foundation, MQTT and AMQP have OASIS. Or at least in most of the cases. Some of them, like the RESTful interface, were defined in Roy Fielding's PhD dissertation and became an absolute web standard. However, in the long run, that can be a disadvantage. Technological progress moves at a very fast pace, and for any protocol to survive, it must be able to adapt to changes. This is a typical (industrial) survival of the fittest scenario.

External companies can become members of the relevant organizations and take part in shaping its direction. That is advantageous for the companies. After all, they will always know beforehand what should be implemented and where the technological wave will go. It is additionally beneficial because they can attract new companies by highlighting that so many have already joined.

Using the technology is free, but shaping it has its price. In 2021, the prices for joining the organizations are between 3,000 USD and 18,000 USD (OPC Foundation) and between 4,100 and 10,500 USD (OASIS). Even though for MQTT and AMQP there is an umbrella organization that encompasses both protocols, the technical circles consist of different member companies. MQTT is mostly supported by IBM, Microsoft, Solace Systems, Software AG, and dc-square, whereas AMQP is supported by VMWare, Microsoft, INETCO Systems, JPMorgan Chase, Deutsche Börse, and Red Hat. OPC UA on the other hand is supported by more than 450 member companies.

# Conclusion

Each of the protocols presented in this paper is unique in its own way. OPC UA is a whole ecosystem, MQTT is very lightweight, AMQP is lightweight and independent of additional layers on top of TCP, and the RESTful interface is well-known on the Internet. Finding the right solution can be a very difficult, if not impossible, task. Each one has its strengths and weaknesses and can show its advantages in a suitable use case.

The more logical and thinkable scenario could be that all of these protocols, in addition to other protocols that will be introduced in the future or are already there but still in the early development phase, will work in parallel to one another in an IIoT scenario. Theoretically, they could be harmonized with each other. They are all TCP-based protocols and adding one layer so that all of them are aware who is on the network and understand each other could be beneficial, especially in the case of pub/sub. However, this is very improbable, because it would mean all organizations coming together and discussing a solution. The most probable outcome will be distributed responsibilities and connectors to the cloud for each one of them.

The possibility exists that one day, after many iterations, there could be one communication protocol that harmonizes everything. Time will tell if this is a realistic scenario. There is also an advantage in the diversity and competition between them. If there was no black, how would a person know what white is? The same is valid here: if there are no competitors, how will it be decided that one protocol is the ultimate solution?

# References


[1] C. Gomez, A. Arcia-Moret, and J. Crowcroft, “TCP in the Internet of Things: From Ostracism to Prominence,” 2018. [Online]. Available: https://ieeexplore.ieee.org/document/8259430/.

[2] IETF, “CoAP (Constrained Application Protocol) over TCP, TLS, and WebSockets,” 2018. [Online]. Available: https://tools.ietf.org/html/rfc8323.

[3] Techopedia, “Challenge-Response Authentication” [Online]. Available: https://www.techopedia.com/definition/26138/challenge-response-authentication.

[4] IANA, “Simple Authentication and Security Layer (SASL) Mechanisms,” 2015. [Online]. Available: https://www.iana.org/assignments/sasl-mechanisms/sasl-mechanisms.xhtml.

[5] R. T. Fielding, Architectural Styles and the Design of Network-based Software Architectures, 2000.

[6] HiveMQ, “MQTT Essentials,” 2015. [Online]. Available: https://www.hivemq.com/mqtt- essentials/.

[7] OASIS, “MQTT Version 5.0,” 2019. [Online]. Available: https://docs.oasis- open.org/mqtt/mqtt/v5.0/mqtt-v5.0.pdf.

[8] OASIS, “OASIS Advanced Message Queuing Protocol (AMQP) Version 1.0,” 2012. [Online]. Available: https://docs.oasis-open.org/amqp/core/v1.0/amqp-core-complete-v1.0.pdf.

[9] OPC Foundation, “OPC Unified Architecture Specification,” 2008. [Online]. Available: https://opcfoundation.org/developer-tools/specifications-unified-architecture.

[10] Wikipedia, “Comparison of MQTT implementations,” 2017. [Online]. Available: https://en.wikipedia.org/wiki/Comparison_of_MQTT_implementations.

[11] X. Chen, “Awesome AMQP 1.0,” 2016. [Online]. Available: https://github.com/xinchen10/awesome-amqp#libraries-and-clients.

[12] M. Zeiss, “List of Open Source OPC UA Implementations,” 2014. [Online]. Available: https://github.com/open62541/open62541/wiki/List-of-Open-Source-OPC-UA-Implementations.

[13] F. Zaninotto, “Awesome REST,” 2015. [Online]. Available: https://github.com/marmelab/awesome-rest.

[14] WolfSSL, “WolfMQTT Client Library,” 2015. [Online]. Available: https://www.wolfssl.com/products/wolfmqtt/.

[15] L. Johansson, “RabbitMQ Best Practices,” 2019. [Online]. Available: https://www.cloudamqp.com/blog/part1-rabbitmq-best-practice.html.

[16] open62541, “An open source implementation of OPC UA” [Online]. Available: https://open62541.org/.

[17] libhttpserver, “C++ library for creating an embedded Rest HTTP server,” 2012. [Online]. Available: https://github.com/etr/libhttpserver.

[18] MQTT, “MQTT Labs” [Online]. Available: https://mqttlab.iotsim.io/.

[19] AMQP, “NetLoad Lab,” 2018. [Online]. Available: https://github.com/Neotys-Labs/AMQP.

[20] R. Abramczyk, “MQTT - a Real-life Use Case Analysis,” 2018. [Online]. Available: https://www.netguru.com/codestories/mqtt-a-real-life-use-case-analysis.

[21] AMQP, “Product and Success Stories” [Online]. Available: https://www.amqp.org/about/examples.

[22] OPC Foundation, “Case Studies” [Online]. Available: https://opcfoundation.org/resources/case-studies/.

[23] Pepperl+Fuchs, “Sensorik 4.0 OpenAPI,” 2021. [Online]. Available: https://app.swaggerhub.com/apis/Pepperl-Fuchs/Sensorik4.0/1.0.0.

# Your automation, our passion.

## Explosion Protection

- Intrinsic Safety Barriers
- Signal Conditioners
- FieldConnex® Fieldbus Infrastructure
- Remote I/O Systems
- Electrical Explosion Protection Equipment
- Purge and Pressurization Systems
- HMI Systems
- Mobile Computing and Communications
- HART Interface Solutions
- Surge Protection
- Wireless Solutions
- Level Measurement

## Industrial Sensors

- Proximity Sensors
- Photoelectric Sensors
- Industrial Vision
- Ultrasonic Sensors
- Rotary Encoders
- Positioning Systems
- Inclination and Acceleration Sensors
- Vibration Sensors
- Industrial Ethernet
- AS-Interface
- IO-Link
- Identification Systems
- Displays and Signal Processing
- Connectivity

**www.pepperl-fuchs.com**

Subject to modifications · © Pepperl+Fuchs · 04/21 · public

**Pepperl+Fuchs Quality**
Download our latest policy here:

**www.pepperl-fuchs.com/quality**

PEPPERL+FUCHS