Multi-protocol bridge generation for M2M communication using MQTT

Marcel Bartholet 1*, Christian Überall 1
1MNI, University of Applied Sciences, Gießen, Hessen, 35390, Germany
*Corresponding author’s e-mail: marcel.bartholet@mni.thm.de

Abstract. MQTT (Message Queuing Telemetry Transport) is known as a quasi-standard for IoT application communication between IoT devices and OPC UA (Open Platform Communication Unified Architecture) is the de facto standard that enables M2M communication in modern industry 4.0. But there are also industrial machines that still use RESTful (Representational State Transfer) APIs as their only communication interface. Integrating machines with these three protocols into automated manufacturing processes and enact a machine to machine communication between them can be challenging. This paper demonstrates how a M2M communication supporting HTTP, MQTT and OPC UA can be implemented using MQTT as the communication protocol. Furthermore, the goal is to create a model for each bridge that can be used to create interface mappings from MQTT to HTTP or from MQTT to OPC UA. This model can then easily be used to generate MQTT clients that bridge the different protocols and enable the M2M communication between machines to share their information.

1. Introduction
Automating manufacturing processes is a big goal for small, medium sized and big manufacturers. This goal has a wide range of benefits which include more productivity, better material use, better product quality, shorter factory lead times and many more. But integrating industrial machines into an automated process is not an easy task, especially with the wide range of available communication standards and protocols.

When automating machines in a manufacturing environment there is a big focus on event-based communication. The information the machines provide is important to control processes, optimize workflows, provide transparency, and much more.

To enable M2M communication between machines with different protocols there are often used programs that understand multiple protocols. But these multi-protocol middleware are often limited to a set of two or sometimes three different protocols. This is due to the fact that the difference between some of them can be quite large.

So instead of introducing multiple middleware platforms into an environment, to try to enable a M2M communication, another possible solution is to introduce little programs that translate one protocol to another and used them wherever needed and introduce one base communication protocol that is used by all machines.

These little translators are often called application layer bridges or just bridges.

The idea of bridging multiple protocols to integrate devices into an environment is not new [16]. There a several proposals especially for IoT protocols. For instance, idea of the QEST broker [1], which is to create a middleware that supports REST and MQTT. Although this project had great
potential, you are often confronted with environments that must consider various other protocols. However, this paper is more about creating bridges for individual devices than replacing a central communication component. In this paper, the focus is primarily on MQTT, REST and OPC UA. Further the focus of the generated mapping here is mainly around the operations to acquire information from machines.

The examples mentioned earlier describe possibilities to bridge one protocol to another, but only few propose a model that represents the data needed to create such a mapping. Derhamy, Rönnholm, Delsing, Eliasson and van Deventer [18] propose an intermediate language based on simple CRUD operations for OPC UA that can be mapped to HTTP and other IoT protocols.

The goal of this work is to show how a mapping from MQTT to both REST and OPC UA could be done and propose a model, which can be used to configure and create bridges to enable M2M communication for machines using these protocols.

1.1. Organisation
The paper is organized as follows. The following section give a brief introduction and into HTTP and MQTT as well as a comparison of both, followed by OPC UA. Section 3 discusses the mapping from MQTT to HTTP followed by section 4 that discusses the mapping from MQTT to OPC UA and section 5 with a conclusion.

2. Background

2.1. Different protocol paradigms
HTTP [2] and MQTT [3] differ a lot regarding their intended usage. HTTP is based on the client/server pattern, where the client sends a request to the server and latter sends a response back to the client. The communication is tightly coupled and synchronous. There are a few approaches to make HTTP more event-based for example the usage of long-polling and webhooks, which are also proposed to be the best choice for pub/sub like approach in [13]. But these two techniques do require a server that offers support for these more event-based messaging approaches. Other than this the only way to get data periodically from a HTTP endpoint is to implement a polling mechanism in the client, which can be a feasible approach sometimes.

MQTT on the other hand is built with the publish/subscribe pattern in mind and offers asynchronous messaging, which further allows to decouple the communication.

With the latest MQTT version 5.0 [4], the standard got a new feature that adds the support for the request/response pattern used by HTTP. This feature is the addition of the response topic paired with the correlation data.

The response topic field can be added to an MQTT message to specify topics to which the response of the receivers of the message should be sent. The correlation data field can be used to identify a particular request-response pair but can be omitted. The response topics can be used without the correlation data.

Another features which is like HTTP, is the addition of custom header fields in the MQTT header. Using this feature, it is possible to define header fields (key-value-pairs) to add metadata information to the MQTT message.

However, these new features introduced in MQTT 5.0 are not mandatory to create a mapping from MQTT to HTTP. It is also possible to implement these features using MQTT version 3.1.

2.2. Defining a main focus for OPC UA
OPC UA is a framework that has a lot more features than MQTT. Its purpose is more to model information of machines and provide services to work with this data rather than just provide a protocol for transmitting data. Although we will use feature that these two have in common which is the publish/subscribe pattern introduces with the OPC UA specification part 14 [13].
The information of an OPC UA server is stored in the information model [9], which is nothing other than a graph of nodes defined in the address space [7] of the server. There are ten service sets [8] with services to manage the address space of an OPC UA server.

There are a lot of different node classes already defined in the standard address space model, but the most important ones in for information retrieval are those which hold the actual information.

The OPC UA framework is a much more complex than MQTT. A complete mapping from MQTT to OPC UA would be too extensive for this work.

The focus here will be to get information from the server’s address space, write and invoke remote procedure calls with function nodes within the server.

3. MQTT to HTTP mapping
After introducing MQTT and HTTP the next step is to look at the protocols to extract the properties needed to build such a mapping. A MQTT client that bridges the information from a MQTT broker to a HTTP server must map the messages from subscriptions to a HTTP endpoint. The corresponding response of an endpoint request then must be published back to the MQTT broker.

3.1. General mapping
The two parts of HTTP requests are the request header and (in most cases) a request body. Since a HTTP endpoint is uniquely defined by its URI and HTTP method, these two properties are essential to create a topic mapping. If the HTTP endpoint is freely definable and not fixed, it would be obvious to create a 1:1 mapping. This mapping could look like as follows:

**Topic:** api/job/start/POST

**HTTP endpoint:** POST /api/job/start HTTP/1.1

But usually the endpoints of machine interfaces are defined by their manufacturers and any mapping must be possible.

The header fields could be either be included within the body of the MQTT message in a separate header object or added to the MQTT message header when using version 5.0.

MQTT v3.1 message:

```json
{
"header": {
 "Content-Type": "application/json",
 "Accept": "application/json",
}
"payload": {
 "programNo": 4771
}
}
```

MQTT v5.0 message:

```json
{
 "programNo": 4771
}
```

Corresponding MQTT header with custom fields:

| Header Field         | Value       |
|----------------------|-------------|
| Content-Type         | application/json |
| Accept               | application/json |

Additionally, to the custom header fields in the MQTT header, version 5.0 introduced an explicit field for the content type. If the payload format indicator of the MQTT header is set, this (optional)
field expects a MIME content type descriptor, but any valid utf-8 encoded string can be used. Alternatively, the content type could also be handled like seen above.

3.2. URL and query parameters

There are obvious parts of many HTTP request URLs that cannot be mapped 1:1 to a topic and that are parameters. One possible solution for this would be to treat them exactly as the fields of the HTTP header. In this case, the order for variables in the URL is important. The query parameters are named anyway and can be included into a separate object of the MQTT message for version 3.1 or the MQTT header for the version 5.0.

HTTP endpoint request:
POST /api/{v1}/machine/{0001}/start?id=user&pw=1234 HTTP/1.1
Content-Type: application/json
Accept: application/json
{"programNo": 4771}

Corresponding MQTT message:
{
"header":
{
"Content-Type": "application/json",
"Accept": "application/json",
},
"queryparams": ["v1","0001"],
"urlparams":
{
"id": "user",
"pw": "1234",
},
"payload":
{
"programNo": 4771
}
}

3.3. HTTP Response handling

In the related work are few to none mentioning about the response data to a corresponding request. We propose two ways to handle the HTTP response to not lose the valuable data from the endpoint.

3.3.1. Using the response topic.

With the use of the response topic feature the response coming from the endpoint can be easily mapped to given topics. The disadvantage of this approach is that the MQTT message must contain the whole data from the HTTP response and the distinction between different status codes and other values would be a part of the subscribing client, which is normal when handling HTTP responses.

200 OK
Content-Type: application/json
{
"state": "started",
"timestamp": 1594394171724
}
3.3.2. HTTP status code mapping.
A more granular approach is to create a switch statement over the HTTP status code. With a mapping from the HTTP status code to a specific topic(s) it is possible to delegate the data of the response more precisely. For instance, it is possible to create a mapping to an explicit topic for a "200 OK" response, so that a client, that only needs to know whether a request to an endpoint was successful, will only receive this required information. Other status codes, that for instance indicate an error, could still be mapped to a specific error topic.

An advantage of this approach is that the topics can be used more precise and it enables a grouping of status codes or a default/fall through case for a topic to which all other response would be sent to. Also, the data transmitted can be reduced by applying a filter to these mappings. In the filter single fields (or the whole body) could be specified to be dropped. Sometimes the event of receiving a message can be information enough. The idea of a filter could also be used for the first suggested option, the response topic.

On the other hand, the topics for the different response codes all need to be specified first. There are over 40 default status codes defined for HTTP/1.1 and it is possible to define custom ones as well.

Figure 1 describes a mapping of a response status code.

![Figure 1: Mqtt2Rest status code mapping](image)

The MachineManager publishes a message to the topic Manager/Machines/0001/Start, which the Rest2Mqtt bridge is subscribed to. On receiving this message, the bridges start a request to the mapped HTTP endpoint. The response is then evaluated and depending on the status code a message is published to the mapped topic.

3.4. Authentication
The HTTP authentication is a part of the HTTP header and could be added to each endpoint individually. To remove this redundancy, a set of common header fields could be defined, which is applied to each individual endpoint that needs the authentication credentials.

But these authentication credentials are not always known at the start. To be more dynamic an extra endpoint could be defined that needs to be called first if the credentials are not supplied. The authentication credentials then could be added to the common header fields.
3.5. Mqtt2Rest model definition

Figure 2 shows a model that holds all the information needed to create a client which acts as a bridge from MQTT to HTTP.

![Mqtt2Rest model diagram](image)

The HttpEndpointDefinition is a representation of the information needed to make a HTTP request to the server. The MqttEndpointDefinition is used to describe a message with its properties and content as well as the topic. Some of the fields are only relevant when publishing a message. An EndpointMapping represents the mapping from an incoming (subscription to a topic) MQTT message to a HTTP request made to the HTTP server. Additionally, the Responses represent a mapping from a predicate applied to the status code of the response to a list of MqttEndpointDefinition to publish the result to. With a present response topic, in the header of the MQTT message, this can be optional.

4. MQTT to OPC UA mapping

A MQTT client that bridges the information between a MQTT broker and an OPC UA server must map the incoming messages from subscriptions to an appropriate action on the server’s address space. The action type can either be put into the body of the MQTT message or added to the MQTT header as metadata. Depending on the action a result must be published back to the MQTT broker.

4.1. Information gathering

The OPC UA service sets, which are responsible for gathering information from the server’s address space, are Attribute for reading and writing data from and to nodes, Method for making remote procedure calls and Subscription and MonitoredItem for creating and managing subscriptions. These are the service sets we will focus our mapping on.
4.1.1. Variable service mapping

The Variable service set offers services to read and write data from and to nodes. To get the value of a node the only thing the bridge needs to know is the node identifier. The message the bridge needs in order to read data from the address space could look like following:

```
Topic: Machines/0002/State
{
    "action": "Read",
    "nodeId": "ns=1;id=1234"
}
```

The action signals a read operation and the nodeId is the unique node identifier in the address space. The received value can then be published on to a specified topic or a list of topics.

To write a value to a node in the OPC UA address space, we need also supply the value that should be written. Otherwise the message is like the one used to read the data.

```
Topic: Machines/0002/Loglevel
{
    "action": "Write",
    "nodeId": "ns=1;id=4711",
    "data": 4
}
```

4.1.2. Subscription service mapping

Creating a subscription to get a value of a variable whenever it changes or get notified through event that happened on the OPC UA server also requires a node identifier. Additionally, an interval can be set as well as a flag to enable or disable the subscription. The bridge subscribes to the specified node in the OPC UA servers address space and gets notified, whenever there is a change of the node. The received value can then be published on to a specified topic. This can either be topics mapped in the data model or response topics set in the MQTT header.

```
Topic: Machines/0002/Progress
{
    "action": "Subscribe",
    "nodeId": "ns=1;id=12345",
    "interval": 5000,
    "enabled": true
}
```

4.1.3. Method service mapping

The last service can be compared to the HTTP request response pattern. Sending a request to an endpoint and compute the result is a lot like making a remote procedure call. The result is not always important and often you do not care for it at all.

The bridge subscribes to an incoming topic with the node identifier and the necessary parameters to call function. The result could then be published as a MQTT message to an outgoing topic. In the case the programming language the bridge is implemented in has problems with the argument type conversion it is possible to pass the argument type information as an additional array to supply information how to convert the values before making the call to the server. Figure 3 illustrates a mapping for a remote procedure call.

```
Topic: Machines/0002/Foo
{
    "action": "Call",
    "nodeId": "ns=1;id=42",
    "args": [1, "4.5", false],
}
```
"argTypes" : ["Int16", "Double", "Boolean"]
}

Figure 3: Mqtt2OpcUa remote procedure call

4.2. Mqtt2OpcUa model definition

Figure 4 shows a possible model that represents all the information needed to create a client which acts as a bridge from MQTT to OPC UA.

The AddressSpaceAction represents the information for the OPC UA client of the bridge. Each action has a node as a target and thus needs a node identifier. The additional fields needed for the different actions are expected to be supplied in the args dictionary. The EndpointMapping differs only by the absence of the status code predicate. Although it is possible to add a general predicate for responses.
4.3. Loose ends
There are lots of OPC UA properties which were not covered here: the security and session management, historical data and especially the node management. For instance, to add connection information for the OPC UA server, the security policy, the message encryption and the user token type also need to be supplied in addition to the endpoint URL [22].

5. Conclusion
Using the approach of generating application layer bridges for single machines where they are needed can be a great advantage over introducing a whole multi-protocol middleware into an environment. But this clearly depends on the environment where machines need to be integrated in and the used protocols and standards.

Using the proposed models to create bridges makes integration of machines into a multi-protocol environment much easier and faster. Even more when using specifications like OpenApi [20] for RESTful APIs or the NodeSet specifications [21] for OPC UA address spaces to translate the data from the specifications into the models. The presented approach offers the possibility to connect machines easily that do not offer standard communications protocols for Industry 4.0 or smart factory, such as OPC UA or MQTT. Especially small and medium size companies can profit from the presented approach. Big companies will be able to connect their “non standardized” machines to the higher systems, like Enterprise Source Planning (ERP) systems or Manufacturing Execution Systems (MES). So, the presented solution helps companies to increase the automation of the entire production process.

References
[1] M. Collina, G. E. Corazza and A. Vanelli-Coralli, "Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST," 2012 IEEE 23rd International Symposium on Personal, Indoor and Mobile Radio Communications - (PIMRC), Sydney, NSW, 2012, pp. 36-41, doi: 10.1109/PIMRC.2012.6362813.
[2] Fielding, R., Ed., and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, https://www.rfc-editor.org/info/rfc7231.
[3] MQTT Version 3.1.1 Plus Errata 01. Edited by Andrew Banks and Rahul Gupta. 10 December 2015. OASIS Standard Incorporating Approved Errata 01. http://docs.oasis-open.org/mqtt/mqtt/v3.1.1-errata01/os/mqtt-v3.1.1-errata01-os-complete.html. Latest version: http://docs.oasis-open.org/mqtt/mqtt/v3.1.1/mqtt-v3.1.1.html.
[4] MQTT Version 5.0. Edited by Andrew Banks, Ed Briggs, Ken Borgendal, and Rahul Gupta. 07 March 2019. OASIS Standard. https://docs.oasis-open.org/mqtt/mqtt/v5.0/os/mqtt-v5.0-os.html. Latest version: https://docs.oasis-open.org/mqtt/mqtt/v5.0/mqtt-v5.0.html.
[5] R. T. Fielding, (2000), Fielding Dissertations: CHAPTER 5: Representational State Transfer (REST), Retrieved from https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm.
[6] OPC unified architecture - Part 1: Overview and concepts, IEC TR 62541-1, 2016.
[7] OPC unified architecture - Part 3: Address Space Model, IEC 62541-3, 2015.
[8] OPC unified architecture - Part 4: Services, IEC 62541-4, 2015.
[9] OPC unified architecture - Part 5: Information Model, IEC 62541-5, 2015.
[10] OPC unified architecture - Part 8: Data Access, IEC 62541-8, 2020.
[11] OPC unified architecture - Part 9: Alarms & Conditions, IEC 62541-9, 2020.
[12] OPC unified architecture - Part 10: Programs, IEC 62541-10, 2015.
[13] OPC unified architecture - Part 14: PubSub, IEC 62541-14, 2020.
[14] P. Peniak, E. Bubeníková and J. Spalek, "Model of Integration Gateway for Communication of
OPC/MQTT Devices," 2020 Cybernetics & Informatics (K&I), Velke Karlovice, Czech Republic, 2020, pp. 1-5, doi: 10.1109/KI48306.2020.9039852.

[15] B. Oniga, A. Munteanu and V. Dadarlat, "Open-source multi-protocol gateway for Internet of Things," 2018 17th RoEduNet Conference: Networking in Education and Research (RoEduNet), Cluj-Napoca, 2018, pp. 1-6, doi: 10.1109/ROEDUNET.2018.8514136.

[16] M. A. A. da Cruz, J. J. P. C. Rodrigues, E. S. Paradello, P. Lorenz, P. Solic and V. H. C. Albuquerque, "A Proposal for Bridging the Message Queuing Telemetry Transport Protocol to HTTP on IoT Solutions," 2018 3rd International Conference on Smart and Sustainable Technologies (SpliTech), Split, 2018, pp. 1-5.

[17] T. Mizuya, M. Okuda and T. Nagao, "A case study of data acquisition from field devices using OPC UA and MQTT," 2017 56th Annual Conference of the Society of Instrument and Control Engineers of Japan (SICE), Kanazawa, 2017, pp. 611-614, doi: 10.23919/SICE.2017.8105594.

[18] H. Derhamy, J. Rönnholm, J. Delsing, J. Eliasson and J. van Deventer, "Protocol interoperability of OPC UA in service oriented architectures," 2017 IEEE 15th International Conference on Industrial Informatics (INDIN), Emden, 2017, pp. 44-50, doi: 10.1109/INDIN.2017.8104744.

[19] E. Al-Masri et al., "Investigating Messaging Protocols for the Internet of Things (IoT)," in IEEE Access, vol. 8, pp. 94880-94911, 2020, doi: 10.1109/ACCESS.2020.2993363.

[20] Darrel Miller, Jeremy Whitlock, Marsh Gardiner, Mike Ralpshon, Ron Ratovsky, Uri Sarid, (2020 February 20), OpenAPI Specification, Retrieved from http://spec.openapis.org/oas/v3.0.3

[21] OPC Foundation, UA-NodeSet, (2016), GitHub repository, https://github.com/OPCFoundation/UA-NodeSet

[22] OPC unified architecture - Part 2: Security Model, IEC 62541-2, 2016