Protocol interoperability of OPC UA in Service Oriented Architectures

Hasan Derhamy, Jesper Rönnholm*, Jerker Delsing, , Jens Eliasson, and Jan van Deventer
Dept. of Computer Science, Electrical and Space Engineering
Luleå University of Technology, Luleå, Sweden.
Email: name.name@ltu.se

*Email: jesrnn@gmail.com

***Abstract*—Industrial Internet of Things covers all aspects of networked intelligent manufacturing systems. This means covering a wide array of application domains and user requirements. In such scenarios it is not feasible to define a single protocol for all situations. Hence, a multi-protocol approach is required. OPC UA has strong backing from Industry 4.0 as the protocol for the Industrial Internet of Things. Interoperability of OPC UA has been investigated in the context of migration from legacy and with protocols such as DPWS. Additionally HTTP and CoAP have been investigated as possible transport mediums.**

However, OPC UA interoperability has not been investigated within a multi-protocol settings and no generic protocol translation exists. This paper proposes an OPC UA translator following the service translator model proposed in the Arrowhead project. Utilizing a mapping to intermediate format, it can be used along side CoAP, HTTP and MQTT protocols.

***Index Terms*—OPC UA, Industrial Internet of Things, Industry 4.0, Protocol Translation, RESTful, Arrowhead Framework**

I. Introduction

Internet of Things (IoT) communication protocols cover highly varied application domains. Trying to select a single communications protocol reduces flexibility and removes possibility for leveraging domain specific benefits of a certain protocol. Therefore, to be able to operate in a multi-protocol ecosystem a form of translation is required. Rather than a dedicated middleware or network proxy, Derhamy et al. propose an "on-demand" translation service. Following Service Oriented Architecture (SOA), this translation service is injected between a service exchange when a mismatch is detected by an Orchestrating entity.

Object linking and embedding for Process Control - Unified Architecture (OPC UA) is an industrial communication framework that is highly promoted for integrating distributed systems. While it has many promising features which we describe in section II, there are still situations where other communication protocols have benefits. Primarily in constrained environments protocols such as Constrained Application Protocol (CoAP) is well suited for battery powered and processor constrained applications. But also, as communications protocols change and improve, solutions must adapt to the new technology. Hence, a successful interoperability solution must work with existing and future protocols.

There is existing work towards migrating applications from existing protocols and frameworks to OPC UA and on integrating OPC UA with other protocols. Some of these works are presented in the next section.

A. Related work

There exist a number of solutions (proposed or implemented) with the aim of increasing factory floor interoperability. A common theme is vertical integration, where solutions for achieving interoperability have involved implementing OPC UA and Device Profile for Web Services (DPWS) to enable high-level access of field-level data [1], [2]. Candido et al. in [1] analyze the two protocols, OPC UA and DPWS, for effectiveness in access device level (field level) data. They concluded that neither protocol could completely satisfy all requirements for Industrial IoT. However the combination offered complementary benefits.

In [2], Sauter et al. describe a protocol gateway used to connect high level IP based networks to field-bus device level data. While maintaining ability to operate with any field bus, they restrict the IP interfaces to a smaller subset available. The gateway is responsible for more than protocol translation as it does logging and historical storage also. They do not however describe what is required to setup such a gateway. Manual configuration and design time consideration of the gateway is described as a necessity for the use of a gateway architecture.

Another solution proposed for enabling interoperability with OPC UA is described in [3]. It again uses a gateway approach with a complex event processor bridging DPWS and OPC UA networks. The gateway uses a middleware to subscribe to events originating from an OPC UA server and from DPWS enabled devices. The events are then translated so that a complex event processor can handle events from both protocols. Here, the middleware must be pre-configured to subscribe to the desired events and the events must be translated for processing.

Arguments for direct modification of OPC UA to better adapt it to the industrial IoT have also been raised by Grüner et al. in [4] and [5]. Specifically, this has involved a series of adaptations to the OPC UA binary protocol, enabling stateless service requests and reducing communication over-

978-1-5386-0837-1/17/$31.00

head - thereby making it more RESTful, and more friendly to resource-constrained devices. Similarly CoAP has been proposed as a transport option for the OPC UA stack [6]. Here, CoAP is the transport messaging supporting the OPC UA services [7]. Therefore, CoAP applications using that solution must implement OPC UA services.

Releases of the initial OPC UA specification included an SOAP/XML/HTTP API. Since release 1.03 this has been deprecated. This was due to a lack of industrial adoption of WS Secure Conversation.

There are already a few proprietary solutions that strive for interoperability involving OPC UA. An example of this is Kepwares KEPServerEX [8], which expose its data as legacy OPC DA data (tags) accessible through OPC UA services. Interoperability is provided by a gateway [9], which also allows third-party access through REST- and MQTT-client APIs.

HyperUA [10] is another proprietary solution for accessing OPC-UA servers from web clients. An HTTP server provides a gateway for web clients to address HyperUA nodes, references, monitored items, servers and subscriptions. The gateway handles sessions to the OPC UA servers and translates Hyper nodes to OPC-UA nodes. In order to interact with an OPC UA server, clients must implement the HyperUA API.

The primary problem with interoperability solutions such as gateways and protocol adapters is that they require custom configuration per site, and do not scale well with each additional protocol added to the mix. As the complexity of a centralized gateway grows, the solution becomes more brittle and resists change.

B. Contribution

OPC UA applications have not been used alongside standard web applications without knowledge of OPC UA. As stated in the related works previous efforts with HTTP and CoAP have been to use them as a transport for regular OPC UA communications.

This paper proposes an OPC UA translator that works with standard IoT protocols such as HTTP, CoAP and MQTT. Enabling access to OPC UA nodes from non-OPC UA based IoT applications. The paper presents:

1) Mapping between a subset of the OPC UA services and an intermediate format which can then be translated to any other IoT protocol.
2) Mapping between OPC UA address space and generic IoT protocols.
3) Implementation demonstrated upon an Industrial use case for condition monitoring.
4) Challenges for complete OPC UA service translation.

The paper, in Section II introduces the OPC UA protocol and high level aspects relevant for translation. Following this, Sections III and IV present the proposed solution and the use case. Finally, a discussion and conclusion are in Sections V and VI.

II. OPC Unified Architecture

OPC UA is a protocol for communication in industrial automation, developed by the OPC Foundation and standardized in IEC-62541 [11]. It defines services through which OPC UA clients interact with information models maintained by the server. The base component of the information model is an object-oriented entity called *node*. A node is identified through a *nodeid* which consists of a namespace index and a node name. Edges between nodes are *references* which create semantic relationships. References enable modeling of both hierarchical tree and horizontal mesh graphs. Every node conforms to a *nodeclass* which specifies the attribute set of the given node. There are two groups of nodeclasses: type classes and access classes. By navigating type information and node relationships, an OPC UA client can access node and reference data to gather its own understanding of any OPC UA information model.

Interactions between an OPC UA client and server is standardized through a set of services provided by the server. The services allow access to and management of nodes: 1) management of node and information model; 2) read and write data to nodes, both query and subscription based; and 3) establishment of communication channels to perform further requests. A server can choose to support only a subset of the full service set, this is determined by using a server profile. OPC UA can be used as a data oriented historian. Clients store data within a defined information model. Upper layer functions can then access the data stored in an asynchronous manner.

In the next section the interoperability solution is proposed and the fundamental architecture described.

III. Proposed Translation Solution

The proposed interoperability solution consists of a multi-protocol translator, which is injected into a service exchange in an on-demand basis. The primary aspects of the translator have been described in previous work [12] and so only briefly mentioned in this paper. The main contribution of this paper is proposing a method of mapping OPC UA to an intermediate format, which can then be mapped to other standard IoT protocols such as CoAP, HTTP and MQTT.

Firstly, interactions are defined by OPC UA services. Standard Create, Read, Update, and Delete (CRUD) operations can be used for translating a subset of these services. Next, the address space in OPC UA is defined by nodes. A URI can be used to uniquely address a node. The mapping has been designed to try to minimize leakage of the OPC UA protocol into the translated protocol.

The next sub-section will describe the base architecture on which the translator is based.

A. Base architecture

The base architecture of the translator as presented by Derhamy et al. in [12], consists of a translation hub that is composed of two protocol spokes. The protocol spokes are based on freely available libraries and handle protocol specific

behaviors. The protocol spokes communicate by an intermediate format, which is protocol agnostic. The intermediate format is intended to preserve the properties of the incoming protocol and reduce information loss when translated to the target, outward, protocol. Figure 1 shows this in a block diagram. In the proposed solution, the OPC UA protocol spoke must handle all OPC UA translation and preserve the properties as much as possible.

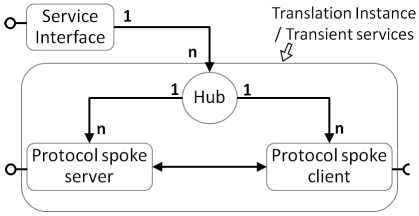

Fig. 1. Block diagram of the base translator architecture

The details of the protocol spoke implementation can be found in Sub-section III-E.

B. Service translation

As introduced in Section II, a subset of the OPC UA services are only used to establish the communications channel, the session and secure channel. These services are completely handled by the OPC UA protocol spoke. This means that they are not exposed to the intermediate format.

The translator's intermediate format uses the RESTful methods to communicate CRUD, namely, GET, POST, PUT, and DELETE. These methods are mapped to the OPC UA services as shown in Table I. This is a mapping from intermediate format to OPC UA services.

TABLE I
OPC UA SERVICE MAPPING TO INTERMEDIATE FORMAT

Operation	RESTful	OPC UA Service
Create	PUT	AddNodes
Read	GET	Read, Browse
Update	POST	Write, AddReferences, DeleteReferences
Delete	DELETE	DeleteNodes

When a *Create* request is passed to the OPC UA spoke, the node id is stored in the URI and the information for creation is stored in the payload. Only the *AddNodes* service is mapped to the Create operation. This is where the OPC UA specific information leaks into the non-OPC UA protocol.

When a *Read* request arrives at the OPC UA spoke, it could be either a read service request or a browse service request. These services are differentiated based on the presence of an attribute index in the URI path. If no attribute index is present, the translator spoke performs a *Browse* on the node.

An *Update* corresponds to 3 services that modify an existing node: *Write*, *AddReferences* and *DeleteReferences*. References are not addressable and belong to a source node. Hence, the translator takes the node id in the URI as the source node, and requires further information in the payload. Both deleting and adding references are treated in the same manner, except that the payload information can differentiate the operation. In order to perform a write operation on a node, an attribute index must be supplied in the URI. The translator addresses the attribute of the node directly. The payload of a write operation is treated as a serialized string, hence it can be JSON, XML, CBOR, EXI, or etc. This is the most common approach for IoT protocols such as CoAP, HTTP, MQTT and XMPP.

A *Delete* operation is translated as a *DeleteNodes* service request. Removing the node corresponding to the node id stored in the URI. The request will also delete all references of the target node.

C. Address space translation

OPC UA address space uses servers, name space indexes, node names and attribute indexes to provide a fine level of granularity to read and write data. These can be broken down into URI format for IoT protocols to construct. In Figure 2 the URL structure is shown with the OPC UA server endpoint defined by the IP address, port and base path. The name space index (ns-idx) and node name (name) make up the complete node-id which uniquely identifies a node on an OPC UA server.

```
<IP>:<port>/<path>/<ns-idx>/<name>/<attr-idx>/<arr-idx>
```

Fig. 2. The general URL structure to address an OPC UA node. Attribute-index can be added for attribute-granularity, and array-index can be added for accessing single elements in arrays stored under attributes.

To reduce the non-OPC UA path parametrization when translating to OPC UA, the URI format uses string for the node name format. The standard node-set, in the OPC Foundation name space, use numeric node names, so for this name space the node names are interpreted as numeric. Hence, nodes with name space index 0 will have their name interpreted as a numeric type.

A reference is identified in a triple: the source node, target node and reference type. The proposed translator addresses references according to a source node described in the URI. The target node and reference type is provided in the payload.

D. Request and Response formats

The translator aims at having least interference in the payload structure or meaning. This is not always possible due to the node structure and service interfaces of OPC UA. The request and response formats are required to follow certain parameters. A non-OPC UA interface must be prepared to provide structured information for management of nodes and references. However, there is no requirement on a write operation, aside from being limited to a serialized string. While the examples are in JSON encoding, XML is just as applicable. JSON has certain advantages over XML, in this case the reduced overhead in the encoding and readability.

When writing to a single node, *update operation*, the request payload format is shown in Figure 3. The node can be accessed down to the variable attribute, which is addressed in the

URI. The value is the only parameter in the payload and can be passed straight through to the OPC UA domain without modification.

```
POST /<namespaceindex>/<nodename>/<attributeindex>/
Payload: JSON Object or XML Document
```

Fig. 3. Write request format for a single node attribute value

When performing a read, the same URI format is used. During a read operation, the payload is not modified, it is serialized as a JSON object or XML document and given to the intermediate format. In the case of browse operation the attribute index is omitted from the URI. Payload was not modified because, ideally, the *protocol translator* should not get involved in semantic translation. A semantic translator would be responsible and could work in tandem with the protocol translator. However full semantic translation is outside the scope of this work.

On the other hand node and reference creation requires additional information in the payload. The structure of a payload is shown in Figure 4.

```
PUT /<namespaceindex>/<nodename>
{"node": {
      "parent": "<parentnode-path>",
      "reference": "<referencetypenode-path>",
      "browsename": "<browsename-string>",
      "nodeclass": "<nodeclass-index>",
      "type": "<typenode-path>",
}}
```

Fig. 4. JSON object format for adding a node.

In order to enable node and reference instantiation a formulation for node- and reference representation was provided for parametrization. Figure 5 shows the add reference payload format, the source node is defined in the URI structure and the reference belongs to this node. The ”forward” parameter specify in what direction non-symmetrical reference will point their semantics. This was one of the areas where OPC UA API must have representation within the payload structure.

```
POST /<namespaceindex>/<nodename>
{"addReference": {
      "type": "<referencetype-node-path>",
      "target": "<targetnode-path>",
      "nodeclass": "<target-nodeclass-index>",
      "forward": "<boolean>"
}}
```

Fig. 5. JSON object format for adding a reference.

Figure 6 has the format for deleting a reference. It follows the same format as the add reference. The difference is in object name, ”deleteReference”, and the lack of the nodeclass parameter, which is only needed when adding references. The deleteReference object is how the translator differentiates between the service calls.

```
POST /<namespaceindex>/<nodename>
{"deleteReference": {
      "type": "<referencetypenode-path>",
      "target": "<targetnode-path>",
      "forward": "<boolean>"
}}
```

Fig. 6. JSON object format for deleting a reference.

In the next section the proof of concept implementation is presented.

E. Proof of concept implementation

The proposed translator has been implemented using Java and applied to the use case presented in Section IV. The protocol spoke uses the OPC Foundation Java client library to perform client specific functions, including creating the session/secure channel and executing the service request to the target OPC UA server. The OPC UA protocol spoke is instantiated by the translation hub and internally routed to a corresponding protocol spoke for the *other* protocol. The session and secure channel is setup on spoke instantiation. The current prototype uses anonymous certificates to connect to the OPC UA server.

Requests from non-OPC UA protocol spokes are processed into the intermediate format and passed to the OPC UA protocol spoke. The protocol spoke parses the URI path and payload body to extract OPC UA node address and any service parameters. These parameters are used to select the OPC UA service to be used to communicate with the server and to construct the OPC UA object payload. The OPC UA client then performs a synchronous service invocation to the OPC-UA server. The response from the OPC-UA server is presented to the protocol spoke as OPC-UA parameter objects. These objects are re-serialized to a JSON string and passed as a response in the intermediate format.

The proof of concept has been tested between a CoAP client and an OPC UA server. This use case is presented in next section.

IV. Application Scenario

The proposed solution can be used as a middleware solution with permanent presence in the communication path. Another approach is to provide the translator as a service which dynamically can be invoked based on need. In the here described application scenario, the translator is a service. The SOA-based Arrowhead Framework [13] enables dynamic provisioning and composition at run time. Leveraging these features the proposed translator can be invoked dynamically based on the current orchestration of service consumption. In this way the Arrowhead Framework supports multi-protocol System of Systems. This may become a key enabler to Industrial Internet of Things.

At the core of the Arrowhead framework are three mandatory core services, these are:

1) ServiceRegistry - provides a store for publishing service provider presence and performing look-up.
2) Service Authorisation - provides authorisation for service consumption.
3) Service Orchestration - provides match making and advanced service look-up.

Arrowhead also defines support core services. These services are used only when needed. The Multi-protocol translator is one of these support core services. The proposed solution extends the existing translator with the OPC UA protocol.

In the context of the proposed solution the Orchestration is responsible for, firstly, identifying a protocol miss-match between a potential service provider and consumer. Secondly, it issues provisioning requests to the translator, to which the translator instantiates two endpoint spokes which match the protocols of each service provider and consumer respectively. The detail is described in Section III-A

The Arrowhead framework documentation architecture defined by Blomstedt et al. in [14] describes three levels: services, systems and systems-of-systems. Of interest to translation; the service layer documentation captures 1) abstract service description, 2) interface design, and 3) protocol and semantic profiles. These artifacts are used to identify the protocol miss-match between matching services.

A pilot scenario from the Arrowhead project has been selected to demonstrate the proposed solution. The scenario involves monitoring the vibrations and heat produced by ball bearings on a Volvo wheel loader. Volvo wheel loaders are expensive machinery which must be maintained in order to extend lifetime. By monitoring the ball bearing it is possible to measure wear and schedule preventative maintenance. In addition, monitoring the rotations of the wheel, it is possible to ascertain the performance of the ball bearing and find early end of life.

In order to extract information at such a granular level from the industrial machinery, IoT technologies offer a promising choice. IoT based ball bearings is more than simply connecting the Wheel loader to the Internet. Industrial environments require a solution which is robust to communication interruption, long service life and able to scale up to many thousands of devices within a single solution. This means, that the solution needs low-power operational modes and intelligent, automated, re-configuration.

Constrained Application Protocol (CoAP) is a preferred protocol for low-power environments, offering support for RESTful interface design [15]. In this use case a CoAP based ball bearing sensor is running a client as a service consumer. It seeks-out a service provider where it is able to push sensory data. Arrowhead orchestration is used to find such a service provider which matches the needs of the ball bearing sensor. The Orchestration has been loaded with a description of the requirement for the sensor. The requirement is such:

1) If available, use the head-office historian,
2) else, use a local historian to temporarily store data.

The Systems of Systems (SoS) Description has a head office historian and a local historian. If Internet connection is available, then the head office historian is used, otherwise the local historian is used. When the Orchestration service is used, it will return either the local or the remote historian. This decision is based on availability, the remote service provider is dependent on an Internet connection. However, the head office historian must integrate with many more systems than the simply the ball bearing sensor. It has been implemented with OPC UA, perhaps according to company policy. Figure 7 shows the SoS with an Internet connection and Figure 8 without an Internet connection.

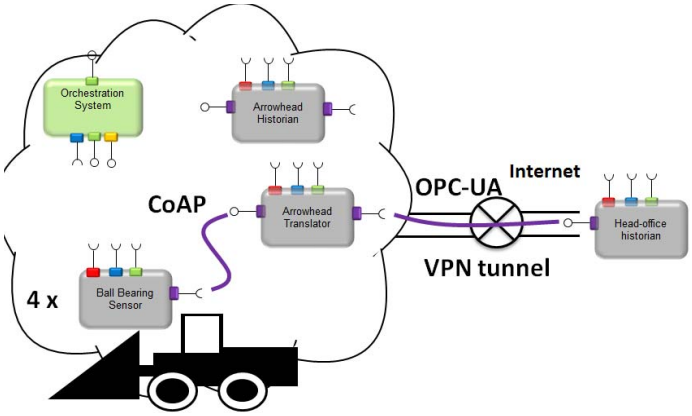

Fig. 7. System of System diagram when Internet connectivity with VPN is possible

In this case we have an interoperability issue between the ball bearing and the remote historian. It would be possible to implement a local gateway which caches the data and forwards to the remote historian. However, this gateway would require implementation, testing and configuration effort. Furthermore, the local historian in this use case supports CoAP. So, in the local connection there is no interoperability issues. The proposed solution is used here as an ”on-demand” protocol bridge when communicating with the head office, while saving resources when direct communication is possible between sensor and local historian.

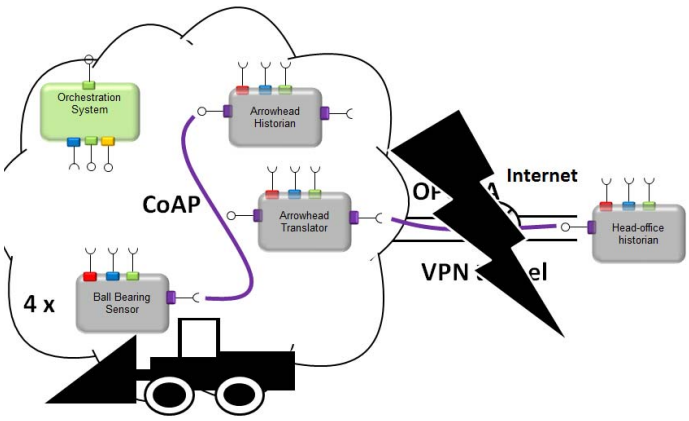

Fig. 8. System of System diagram when no Internet connectivity is possible

The application scenario is setup as shown in Figure 9. The sensor is implemented on an embedded processor, the Mulle

[16]. It connects to an Arrowhead local cloud running on a BeagleBone Black (BBB). The OPC UA server is running an a general purpose PC with an OpenVPN connection to the BBB.

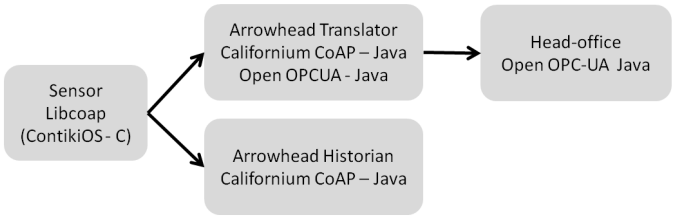

Fig. 9. Demo technology usage and setup

The interactions between sensor, Arrowhead systems is following pre-defined Arrowhead patterns [13]. The OPC UA historian information model is shown in Figure 10. The interface is to a single OPC UA node for each ball bearing. The node is of type variable and stores a string representation of the JSON object sent from the ball bearing.

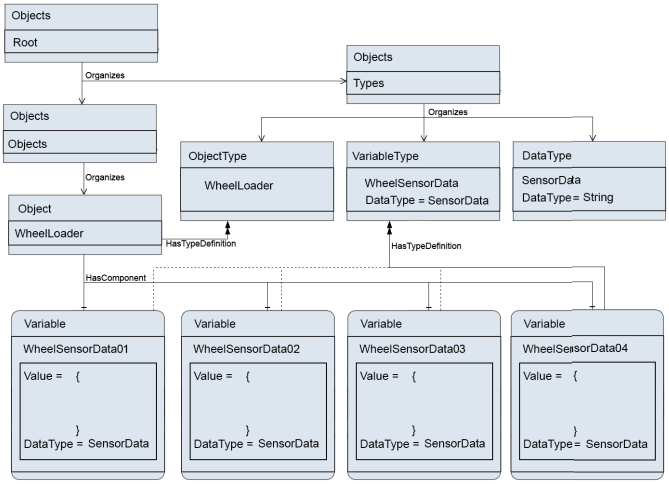

Fig. 10. OPC UA historian information model

The sensor uses a CoAP POST operation to push data to the historian service provider. As discussed in Section III this is translated to an OPC UA node write operation. The translator maps OPC UA write operations to a fully addressed node-id and attribute index, therefore no additional OPC-UA semantics leaks into the payload. This is shown in Figure 11. Here, rotation, temperature and min/max/rms vibration data is stored as a SenML JSON string. The name space index is 5, the node name is wheelsensordata0x and the attribute is 13. The node name changes for each ball bearing.

This orchestrated historian enables end-to-end communication, with resilience to Internet disruption, and flexibility between communication protocols. The proposed implementation allows service based systems to push their data to an OPC UA server, regardless of their protocol usage. In the following section, the proposed solution will be discussed.

```
POST /5/wheelsensordata0x/13/ CoAP
Host: 192.168.7.1:48010
{
    "e":[
        {"n":"rotation", "v":1, "u":"count"},
        {"n":"temperature", "v":60, "u":"Cel"},
        {"n":"vibration_max", "v":4, "u":"m/s2"}
        {"n":"vibration_min", "v":0, "u":"m/s2"}
        {"n":"vibration_rms", "v":3, "u":"m/s2"}
    ],
    "bn":"urn:dev:mac:0024befffe804ff1"
}
```

Fig. 11. CoAP payload example and structure, three nodes being updated in a single CoAP message

V. Discussion

The translator has attempted to reduce leakage of OPC UA semantics across to native web protocols. As with a translation between MQTT and HTTP where a path in HTTP represents a topic in MQTT, a path in HTTP represents a node in OPC UA. However the path structure must conform to a particular structure. By utilizing common methods between HTTP and CoAP with OPC UA, the OPC UA protocol spoke does not require additional semantic information within the non-OPC UA interface definition.

In this translation, the core features of the OPC UA information model have been preserved. Namely, it is possible to address individual nodes and to browse nodes. Therefore it is possible to explore the address space and to read and write to nodes.

However, OPC UA supports 37 different services while the proposed solution has only mapped 7 services. The supported service set provides a bare minimum interoperability for non-OPC UA systems to interact with OPC UA servers. In this way payload interface requirements made on the non-OPC UA systems is kept to a minimum. In fact for read and write, only the addressing scheme of the service interface is impacted This impact is also kept to a RESTful style, thereby would not be unnatural to any REST API developer. It could be possible to include the full set of OPC UA services within the URL, as path variables or URL query parameters. This would however break RESTful design approach by including operation information in the address space. It would also apply further interface design considerations on the non-OPC UA systems.

A service, in SOA concept, provides a functionality. This is beyond the interaction services defined in OPC-UA. A call made to a service is expected to result in some value added result. Perhaps the concept of the method object in OPC UA could map well to a functional service. So an OPC UA server would provide services such as ”sound fire alarm”, ”turn off lights” or ”store wheel loader data”. It is these services which should be registered in a generic service discovery methodology such as DNS-SD would allow all systems (OPC UA or not) to discover services provided by an OPC UA server.

VI. Conclusion

Communication in Industry 4.0 requires flexible interconnections between integration layers and information layers. The OPC UA communication framework shows potential for such interconnections. Convergence to a single protocol is possible, but it is likely that a multi-protocol communication network will exist for some time.

The here proposed translation service enables interoperability between OPC-UA and other SOA protocols supporting integration of legacy automation system with upcoming IoT devices. Such integration is supported by integration platforms like the Arrowhead Framework. Frameworks such as Arrowhead enable dynamic provisioning and composition of the solution into the service exchange communication path. The proposed solution allows for OPC UA translation mapping to be defined once, and then used for CoAP, HTTP and MQTT protocols.

The translation preserves management, browsing and read-/write of OPC UA nodes. Because of the complexity of the OPC UA service interface, the translation does not cover all functions of OPC UA. The mapping from OPC UA address space, requires that nodes are addressed in the URL path or topic name in a specific manner. The name space index and node name in the path are required to address the node. Combining this with standard CRUD methods, a subset of OPC UA services can be invoke from a generic non-OPC UA service interface.

VII. Future work

Future work involves working translation back from OPC UA to HTTP/CoAP/MQTT. This would enable OPC UA clients access to non-OPC UA services.

Incorporating OPC UA server discovery into a generic service discovery is also required for full look-up and runtime binding routines to be performed.

Investigating the possibility of expanding the supported OPC UA service set while maintaining a minimum requirement on service interfaces implemented with non-OPC UA protocols.

An interesting new protocol for IoT is gRPC. Developed by Google and used within their microservices infrastructure, it would be interesting to investigate its suitability for interoperable use with OPC UA. It would enable cloud based service interaction with automation services on the factory floor.

Acknowledgment

This work is supported by the EU ARTEMIS JU funding, within project ARTEMIS/0001/2012, JU grant nr. 332987 (Arrowhead).

References

[1] G. Cândido, F. Jammes, J. B. de Oliveira, and A. W. Colombo, "Soa at device level in the industrial domain: Assessment of opc ua and dpws specifications," in *2010 8th IEEE International Conference on Industrial Informatics*, July 2010, pp. 598–603.

[2] T. Sauter and M. Lobashov, "How to access factory floor information using internet technologies and gateways," *IEEE Transactions on Industrial Informatics*, vol. 7, no. 4, pp. 699–712, Nov 2011.

[3] M. J. A. G. Izaguirre, A. Lobov, and J. L. M. Lastra, "Opc-ua and dpws interoperability for factory floor monitoring using complex event processing," in *2011 9th IEEE International Conference on Industrial Informatics*, July 2011, pp. 205–211.

[4] S. Grüner, J. Pfrommer, and F. Palm, "A restful extension of opc ua," in *2015 IEEE World Conference on Factory Communication Systems (WFCS)*, May 2015, pp. 1–4.

[5] ——, "Restful industrial communication with opc ua," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, pp. 1832–1841, Oct 2016.

[6] P. Wang, C. Pu, and H. Wang. (2017) Opc ua message transmission method over coap 01. [Online]. Available: https://tools.ietf.org/html/draft-wang-core-opcua-transmission-01

[7] ——. (2016) Requirement analysis for opc ua over coap. [Online]. Available: https://tools.ietf.org/html/draft-wang-core-opcua-transmition-requirements-00

[8] Kepware. (2017, May) Kepserverex. [Online]. Available: https://www.kepware.com/en-us/products/kepserverex/documents/kepserverex-manual.pdf

[9] ——. (2017, May) Kepware iot gateway. [Online]. Available: https://www.kepware.com/en-us/products/kepserverex/advanced-plug-ins/iot-gateway/documents/iot-gateway-manual.pdf

[10] Projexsys. (2017) Hyperua. [Online]. Available: http://projexsys.com/hyperua/

[11] IEC-62541, *OPC UA Specification*. OPC UA Specification Release 1.03, 2015.

[12] H. Derhamy, J. Eliasson, and J. Delsing, "Iot interoperability - on-demand and low latency transparent multi-protocol translator," *IEEE Internet of Things Journal*, 2016, accepted.

[13] J. Delsing, *IoT Automation Arrowhead Framework*. CRC Press, 2017.

[14] F. Blomstedt, L. L. Ferreira, M. Klisics, C. Chrysoulas, I. M. de Soria, B. Morin, A. Zabasta, J. Eliasson, M. Johansson, and P. Varga, "The arrowhead approach for soa application development and documentation," in *IECON 2014 - 40th Annual Conference of the IEEE Industrial Electronics Society*, Oct 2014, pp. 2631–2637.

[15] Z. Shelby, K. Hartke, and C. Bormann, "The constrained application protocol (coap)," Internet Requests for Comments, RFC Editor, RFC 7252, June 2014, http://www.rfc-editor.org/rfc/rfc7252.txt. [Online]. Available: http://www.rfc-editor.org/rfc/rfc7252.txt

[16] "Eistec AB," April 2017. [Online]. Available: http://www.eistec.se/