REST based OPC UA for the IIoT

Rainer Schiekofer[1,2], Andreas Scholz[1], and Michael Weyrich[2]
[1]Siemens AG [2]University of Stuttgart

***Abstract*—OPC UA is one of the most important communication protocols for Industry 4.0 applications in the automation domain. However, to really reach the status of an Internet-of-Things protocol, OPC UA also has to give some answers to upcoming cross-domain questions. Only if OPC UA is able to bridge the gap between several different domains, it will be accepted as real Internet-of-Things protocol and therefore supported even better by the APIs of the big IT players.**

One technology which already interconnects nearly every domain is the REST architecture. A lot of research already focused on the combination of these two technologies. However, in this paper we will show the first approach which covers all OPC UA requirements and therefore enables also use cases like server reconfiguration. We will also give some insights into our concept, which was contributed to the OPC UA specification as part of the V1.04 release.

I. Introduction

Today our society stands on the edge of a new era. As the processing power and the connectivity of industrial embedded devices increase more and more, a lot of new applications become feasible. This phenomenon has different names in different domains, for example, SmartGrid, SmartHome, Internet-of-Things and of course the nowadays term for the automation domain, Industry 4.0. It is possible that these new properties of industrial embedded devices will change our lives in a similar way as the introduction of smartphones. The success of these new applications depends mainly on the interoperability of the transport layer, to enable different devices to communicate with each other and on the interoperability on the semantic layer, to enable different devices to understand the meaning of the communication. Both of these requirements can be considered enabler for typical Industry 4.0 scenarios.

One of the challenges for addressing interoperability on the transport layer is the huge number of different protocols. Nearly every domain has developed their own protocol to solve similar problems. In the end, we have to answer the question, if we really need a dedicated protocol for each domain, or if we can use a single protocol to address most of the common use cases. With that in mind, we started to look for the most promising Internet-of-Things protocol of the automation domain, which seems to be OPC UA [1], [2]. OPC UA [3] does not only aim to solve the interoperability on the transport layer, instead, also the interoperability on the semantic layer shall be addressed by OPC UA, through the introduction of so-called companion specifications. However, to reach the level of a real Internet-of-Things protocol, OPC UA must be able to reach out to other domains too.

A communication technology, which everybody already knows and also is present in nearly every domain, is the REST architecture [4]. REST is derived from the classic web, which is already connected to each domain in one way or another. The basic idea of this work is to extend OPC UA with REST capabilities to finally reach the status of an Internet-of-Things protocol. A lot of research was already done on this topic [5], [6], [7], [8]. However, our proposal is the first proposal which takes a certain property of OPC UA into account, which is necessary to support reconfiguration scenarios in combination with session-less clients. Without this feature a session-less client is not able to use services like *Read* or *Browse* safely in dynamic environments. Additionally, some parts of our proposal were contributed to the OPC UA specification as part of the V1.04 release.

II. Background

In this Section, we give a short overview of the REST-architecture and OPC UA.

A. Representational State Transfer

Representational State Transfer (REST) was introduced by Fielding in [4]. In his work Fielding derives the REST architectural style from the very successful web. In the end, he formalized a couple of rules for his RESTful architecture:

Client-Server: Andrews et al. [9] defined a server as a process which handles repeatedly requests from clients. From that point of view, a client triggers actions on a server and therefore can be considered the active part, while the server is the passive part and reacts based on the client requests.

Cache: A cache is a component which is placed between client and server and is able to serve previously cached responses from the server for identical requests.

Uniform interface: Fielding et al. [4] stated out, that the uniform interface is the central feature which distinguishes REST from other network-based styles. The four defined constraints for the REST interface are: the identification of resources; manipulation of resources based on representations; self-descriptive requests and responses; hypermedia as the engine of application state (HATEOAS).

Layered system: Garlan et al. [10] defined layered systems as hierarchically organized systems, were each layer provides a service to the layer above and uses services from the layer below.

Statelessness: Statelessness in the context of client-server interaction means, that no session state is allowed. To be more concrete, a client can not take advantage from previously stored information at the server, like, for example, some

978-1-5386-7108-5/18/$31.00 ©2018 IEEE

language settings, instead, all necessary information to process the request must be included in each message.

Nevertheless, even with the rules clearly stated out in his PhD-thesis, a lot of so-called "RESTful" APIs did not fulfill the REST requirements in the eyes of Fielding [11]. One of the core ideas from REST, hypermedia as the engine of application state (HATEOAS), is often overlooked. In a nutshell, the basic idea behind this concept is, that all necessary information to interact with the service has to be part of the resource representation.

B. OPC Unified Architecture

OPC UA [3] is an industrial standard which basically aims to solve the two topmost problems of typical Industry 4.0 scenarios, which are interoperability on the transport and semantic layer.

Interoperability on the transport layer is reached through standardization of different transport protocols like OPC TCP and HTTP(S) combined with various encodings like OPC Binary, OPC XML, and OPC JSON. In the newest release of OPC UA also the well-known publish-subscribe pattern was introduced, which enables cloud connectivity based on transport protocols like AMQP or MQTT. Additional to the description how messages have to be sent over the wire, the OPC Foundation also specified a couple of *Service Sets*, to interact with the data model of OPC UA. Examples of such services are the *Read* service, for accessing OPC UA information, the *Write* service, for storing OPC UA information and a couple of useful other services.

Interoperability on the semantic layer was addressed by the graph-based data model of OPC UA. OPC UA introduces a set of different *Nodes* (categorized in so-called *NodeClasses*). For example, the *Variable NodeClass* could be used to represent some sensor data (e.g., the actual temperature value), while the *Method NodeClass* shall be used to represent some functions of the sensor (e.g., calibrate). These different *Nodes* can be interconnected with so-called *References*. Several *ReferenceTypes* were specified by the OPC Foundation (e.g., *HasSubtype*, *HasComponent* and *HasProperty*) to cover a broad range of use cases. However, it is also allowed to introduce new *ReferenceTypes*, if necessary. In the end, this concept allows to model arbitrary information and therefore could be used by different application domains (e.g., factory automation, building automation, healthcare, mobility, ...) [8], [12] to express domain-specific knowledge.

III. Problem Statement and Goals

Having identified OPC UA as the most promising protocol for Internet-of-Things applications and the REST architecture as a very interesting technology for cross-domain interaction, in this Section, we further analyze the combination of both technologies and after that take a closer look at the session concept of OPC UA. Based on our findings, we then formulate a number of requirements and discuss the basic approach how we want to achieve them.

A. Analysis

As already explained in Section II-A REST has five mandatory rules. We will now take a closer look if all of these rules are already fulfilled by OPC UA, or if some changes have to be made in the OPC UA specification.

Client-Server: As stated out in Part 1 of the OPC UA specification, the basic architecture of OPC UA is based on the client-server pattern.

Cache: Support of caching in the classic web is typically done by special response headers, which allow intermediary servers to determine if the response could be served again for an identical request. However, the typical OPC UA use case is to serve data from field devices. This data often changes on a millisecond base. Because of that, the OPC Foundation came up with a different caching concept, compared to the classic web. In OPC UA the client is able to specify a so-called "maxAge" parameter for the *Read* service. A server now is able to serve a cached sensor value to the client, as long as the timestamp of the latest cached value is in the requested client range.

Uniform interface: Each OPC UA *Node* can be seen as a resource. *Node* representations can be constructed in such a way that the manipulation of resources could be done based on the representation. Self-descriptive requests can be achieved by introducing a way to express the kind of message (e.g., based on HTTP headers like the Content-Type header). The HATEOAS requirement can be achieved through the mapping of OPC UA *References* to the hypermedia concept of links.

Layered system: The typical application domain of OPC UA consists of several layers, for example, ShopFloor-layer, MES-layer, and ERP-layer. Because of that, the layered system architecture is also part of the OPC UA architecture.

Statelessness: The last mandatory requirement of REST cannot be fulfilled with OPC UA V1.03 and earlier because a client always has to establish a session to access the information model of an OPC UA server. This is even true for services like *Read* or *Browse*. A session in OPC UA is used to store some information, for example, the authorization information and the requested locales. However, it is not enough to just identify all information, which is stored at the client/server during the session set up, it is also necessary to check what kind of additional concepts in OPC UA depend on sessions.

B. OPC UA sessions

As identified in the previous section, the OPC UA session concept is the final challenge which stands between OPC UA and the REST architecture. A session has two major responsibilities in OPC UA: first, storing some information on client and server. Second, guaranteeing the consistency between the client and server state. To identify all information which is part of the first category, we have to check which kind of information is transferred only once during the session set up, for example, the locales. However, identifying all information of the second category is a little bit more challenging. For that, we have to analyze what kind of information is only guaranteed to be stable within a session. However, before we can

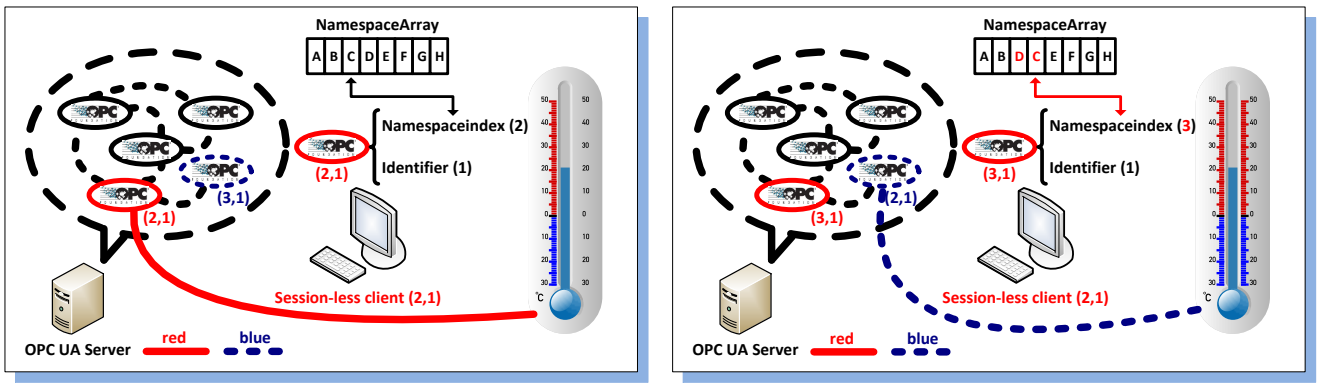

Figure 1. Consequences of a changed *NamespaceArray* during runtime, without further preparations.

address the second category, we have to introduce some OPC UA concept first. The so-called *NamespaceArray* in OPC UA contains an array of URIs. In a nutshell, the *NamespaceArray* is some kind of lookup table, which is used to replace long URIs trough small indices. Based on this concept, *NodeIds* and *ExpandedNodeIds* only contain the corresponding index value, also called *NamespaceIndex* in OPC UA, for the URI, instead of a long URI. For example, the *NodeId* with *NamespaceIndex* 2 (see also Figure 1 left side) refers to the third element of the *NamespaceArray* (e.g., "http://opcfoundation.org/UA/DI"). Part 5 defines the *NamespaceArray* as dynamic, which means, that the content of this array can be changed during runtime (see also Figure 1). *NodeIds* and *ExpandedNodeIds* are used in most of the OPC UA services (e.g., *Read* / *Write* / *Browse* / ...) and because of that the content of the *NamespaceArray* has to be cached during the start up phase by each client. Stale client caches and the fact that changes of the *NamespaceArray* during runtime are explicitly allowed, lead to the problem of Figure 1. The example session-less client wants to display the temperature based on the red *Node*, but fetches the value of the blue *Node*, after the *NamespaceArray* was changed. Of course, such a behavior can not be tolerated for any application. Therefore Part 5 also restricts the way in which these arrays can be altered. For example, a server is not allowed to change these arrays in such a way during an active client session. If a client has an open session, the server can only add some new entries but is not allowed to alter existing entries. But of course, such changes can be done during a reconfiguration, or a reboot, if all sessions are terminated during the process. If the session of a session-based client is terminated, also the cached values for the *Namespace-* and the *ServerArray* are not valid any longer and have to be re-fetched if a new session is established. The problem here is of course, that a session-less client cannot be sure if such a change happens during two subsequent calls.

C. Goals and challenges

In the following Section, we will give some insights into our design goals and the reasons why we want to achieve them.

Efficient SessionlessInvoke: We have already discussed, that OPC UA services which depend on *NodeIds* and therefore on the *Namespace-* and *ServerArray*, need some further investigation. One goal here is to introduce some new efficient service for session-less clients.

Uniform HTTP interface: For our approach we have decided to use HTTP as the underlying protocol, mainly because of huge client support (web browser / web server). To be compliant to the REST-rules, we have to map OPC UA to the uniform HTTP interface.

Batch support: Most OPC UA service sets already implement batch support. This feature significantly reduces the overhead for clients and servers, if more than one value should be fetched (the typical industry use case).

URI templates [13] are used to offer the client a recipe for URI construction. For example, the URI template "https://host/{userName}{?password}" could be used to construct the following URI: "https://host/user?password=secret", where the "userName" is "user" and the "password" is "secret". Besides the usefulness of such a concept for RESTful services, there is a lot of discussion out there whether URI templates should be specified within some kind of document, or should only be discoverable during runtime [14]. However, there are several reasons, why we think harmonizing URI templates across OPC UA servers has more benefits than drawbacks. For example, most actual OPC UA applications know exactly the *NodeIds* which they want to fetch from the OPC UA server (e.g., the well-known *NamespaceArray*). These *NodeIds* are sometimes deeply nested in the information model. The classical hypermedia approach, with only a single well-known entry-point, would require that several "Browse"-Calls had to be made, which would yield in a lot of overhead and can be considered one of the major drawbacks of the HATEOAS approach [15]. Another reason is, that a typical *ExpandedNodeId* OPC UA structure, which is the only way to link to server external *Nodes*, could not be used as link relation to another server if we would not specify a common URI template. Because of that, we think that the benefits of harmonizing URI templates across servers outweigh the drawbacks, in case of OPC UA, and are also applied in the same way by a lot of very popular web services like Amazon's S3 [16], Google's Gmail [17], Apple's News API [18] and the

Twitter API [19].

Browser support: One of the greatest benefits of the classic web is, that we do not have to install a special application for each web-application any longer. Instead, we only have to install one application, also known as a web browser, and can use it for a lot of different applications like webmail, online banking, shopping and a lot more. It is highly possible that this is one of the main reasons, why nearly every device is shipped with a web browser or a web server on board. This is also the main reason, why our REST-binding should also be compliant to a simple web browser, similar to the Amazon S3 REST API [16]. In the end, this would enable the use case to check a single OPC UA value without installing additional applications on most devices. It would be also much easier to download the device documentation directly from the device's OPC UA server by simply tipping in a URL into a standard web browser.

Programming against the TypeDefinition is a concept in OPC UA, which allows someone to find *Nodes* based on their *BrowsePath*. This is, for example, the preferred way to identify *Instance-NodeIds*, based on their complex *TypeDefinitions*. For this use case the *TranslateBrowsePathsToNodeIds* service was introduced by the OPC Foundation. In a nutshell, this service allows you to chain several *Browse* requests in one *TranslateBrowsePathsToNodeIds* request. If we also introduce a common URI template for that service, it would even be possible to apply this concept, in combination with redirections, across several servers.

Resolution of ExpandedNodeIds: The last goal is to be able to resolve *ExpandedNodeIds* in a simple way. This approach also depends on URI templates, because otherwise, it would not be possible to construct a valid URL only based on the actual available OPC UA structures.

IV. Approach

We will now focus on the most interesting parts of our mapping, including an efficient concept for introducing session-less requests in OPC UA, followed by our HTTP mapping. Additionally, we will show how RESTful batch requests can be supported without URL size limitations and also give some insights on how we achieve browser support. Last but not least, we will present a concept for session-less subscriptions.

A. SessionlessInvoke

As already explained in Section III-A, OPC UA can not be considered stateless, even for services like *Read* or *Browse*. This is mainly because of the fact, that the *Namespace*- and *ServerArray* are only guaranteed to be stable within a session.

In Table I the service signature of the *SessionlessInvoke* service is shown. The (+) marks our contributions to the standardized service. There are two ways a client is able to use *SessionlessInvoke* in OPC UA. One way is to specify the used *namespaceUris* and *serverUris* for each call and the other way is to set a so-called *urisVersion* in each call. The idea behind the first approach is straightforward and will not be further discussed here. However, the second approach needs some further explanation. The basic idea behind the *urisVersion* is to versionize the *Namespace*- and *ServerArray*. Every time one of these arrays is changed, also the *urisVersion* must be altered. The *urisVersion* is therefore introduced as new *Property* of the OPC UA *ServerObject*. Of course, a server has to ensure consistency between the *UrisVersion Property* and the *Namespace*- and *ServerArray*. This, for example, could be ensured by using the same semaphore. With that in mind, the concept is also straightforward. At the beginning, a client fetches the *NamespaceArray*, *ServerArray* and the corresponding *UrisVersion*. After that, a client assigns the *NamespaceIndices* based on the cached arrays and sets the *urisVersion* field to the also cached *UrisVersion* value. An OPC UA server now only has to check if the *SessionlessInvoke urisVersion* field matches the local *UrisVersion Property* of the server. If this is not the case because, for example, the *NamespaceArray* was changed in the meantime, the server discards the request and informs the client about the stale cache values with a *BAD_VersionTimeInvalid StatusCode*. After that, a client has to refresh its cache and then is able to retry the request. To ensure cache consistency on client-side, the client should first fetch the *UrisVersion Property* in a single request and after the response is received, try to fetch both arrays. This is necessary because in general there is no transaction context and also no sequential execution guarantee between different items in a batch request. However, it might be possible that some OPC UA servers support this feature within a single batch request, but this should not be assumed in general. Besides the above mentioned contributions, we also contributed some more fixes to support *SessionlessInvoke*, like the redefinition of the *Method* description: "Each *Method* is invoked within the context of an existing session" (OPC UA Part 3 V1.03).

Name	Type	Name	Type
Request		**Response**	
(+) urisVersion	VersionTime	namespaceUris[]	String
namespaceUris[]	String	serverUris[]	String
serverUris[]	String	serviceId	UInt32
(+) localeIds[]	LocaleId	body	*
serviceId	UInt32		
body	*		

Table I
SessionlessInvoke Service Parameters [3].

B. HTTP mapping

A big difference between our proposal and the actual released OPC UA specification is how *SessionlessInvoke* should be mapped to the HTTP protocol. While the OPC Foundation favors an HTTP RPC approach, in which every action is mapped to the POST method, we will show another more RESTful approach.

Table II shows the basic approach how some of the abstract services from OPC UA can be mapped to HTTP. Of course, it is not enough to only specify how OPC UA services should be mapped to HTTP, instead, one of the main topics of each REST-binding is the description of the resource representations. The services which are marked with (+) were introduced

HTTP Method	OPC UA Service	Representation MIME-Type
GET	Read	app/opcua.Boolean+json app/pdf ...
GET	HistoryRead	app/opcua.HistoryReadResult+json
PUT	Write	app/opcua.Boolean+json app/pdf ...
PATCH	HistoryUpdate	app/opcua.HistoryUpdateResult+json app/json-patch+json
GET	Browse	app/opcua.NodeRepresentation+json
GET	BrowseNext	app/opcua.NodeRepresentation+json
GET	TranslateBrowse PathsToNodeIds	app/opcua.BrowsePathResult+json
GET	(+) ResolvePath	app/opcua.NodeRepresentation+json
POST	Call	app/opcua.CallRequest+json app/opcua.CallResult+json
POST	AddNode	app/opcua.CallRequest+json app/opcua.CallResult+json
DELETE	DeleteNode	app/opcua.StatusCode+json
PATCH	(+) Modify References	app/json-patch+json app/opcua.ModifyRefsResponse+json
POST	Query	app/opcua.CallRequest+json app/opcua.CallResult+json
GET	QueryNext	app/opcua.QueryNextResponse+json

Table II
HTTP MAPPING DETAILS (APP = APPLICATION).

in addition to the already existing services. However, these services can be considered as an orchestration of OPC UA services and because of that, the implementation effort is very low.

C. Batch support

Most OPC UA services support batch requests. This feature allows specifying more than one item per service request. The main reason for this feature is to reduce the data on the wire and the necessary processing overhead in client and server applications. Classic REST APIs are often optimized to access single resources. This is mainly because of the different use cases. For example, it often makes no sense to request all webpages of a certain domain or all objects from a given Amazon S3 bucket. Another reason is, that big web services are hosted by a lot of servers, often with caches and load balancers in front of them. Because of that, it would even make more sense to break down batch requests into single requests and distribute them to different servers. In contrast, a typical OPC UA server runs on an embedded controller, which is often the only source for the data. Having identified the necessity of batch requests, we now propose several concepts, how batch requests can be mapped to our RESTful OPC UA API:

1) Definition of some kind of batch-*Node*, which can be configured through the client by adding special *References*.
2) Definition of a *Method*, similar to the classic OPC UA *Read* service.
3) Definition of a special batch URL for receiving and processing concatenate REST requests, similar as depicted in Listing 1.

While the first two approaches look more natural to an OPC UA client, the third one may look more familiar to typical web-clients. However, the second approach should be the approach with the lowest implementation effort. In the end, all three approaches can be used to introduce batch capabilities in a RESTful OPC UA API. For our prototype, we implemented the second approach because this was, besides the lowest implementation effort, also a chance to show some *Method NodeClass* related concepts of our design.

```
[
  {
    "href": "/i=84/BrowseName",
    "method": "get"
  },
  {
    "href": "/i=2255/Value",
    "query": {"timestampsToReturn":2},
    "method": "get"
  }
]
```

Listing 1. OPC UA RESTful batch request (inspired by [20])

D. Browser support

One major reason to use HTTP was the chance to make OPC UA compatible to a standard web browser, without forcing a user to install additional plugins, or placing some gateway server in front of the OPC UA server. This is, of course, no requirement for a RESTful architecture, but might come in handy for use cases like documentation download via a web browser. It is also possible to use this feature for delivering a full-featured OPC UA client based on JavaScript, to gain full access to all features of OPC UA, without installing any plugin (also known under the term code-on-demand).

To enable this feature in the first place, we have to take a closer look at the specification of our *Read* service, which will be used to fetch the *Value Attribute*. If a URL is tipped into the browser an HTTP GET request is sent to the specified URL. However, only mapping the *Read* service to HTTP GET would not do the trick. A browser also interprets the Content-Type HTTP header. If a browser does not know the format, which is often the case for MIME-Types like "application/opcua+uajson", of course, nothing useful can be displayed in the browser window. Because of that, we also introduced a new optional *Property* for *DataType-Nodes* with the *BrowseName* "MIMEType", containing a string value with the MIME-Type (e.g., "application/pdf"), which shall be used as Content-Type HTTP header. But again, this is not enough because some native OPC UA clients will only understand the MIME-Type "application/opcua+uabinary". To also cover this use case an OPC UA server shall interpret the accept header of the request message. If a client specifies some well-known OPC UA MIME-Type as the highest priority, the specified type shall be used to encode the message body. If not, the MIME-Type of the MIME-Type-*Property* shall be returned. Based on the above rules an OPC UA server is now able to deliver, for example, PDFs, which can be directly displayed in the browser, without any additional OPC UA specific plugin.

Another useful optional feature is the possibility to encode all *RequestHeader* fields as HTTP query parameters. Of course, it also makes sense to be able to encode the fields in

HTTP headers, like the OPC Foundation did for the HTTP authorization header (see also OPC UA Part 6 for further information). However, if somebody wants to share a link to an OPC UA *Node*, it must also be possible to encode this token in the URL, otherwise, there is no guarantee, that the value can be fetched in each case. Just consider a simple dashboard web-application, which only allows specifying a URL, but does not allow to set any kind of header like [21].

E. Subscriptions

Services like *RegisterNodes* / *AddSubscription* and *AddMonitoredItems* are not possible with the currently released *SessionlessInvoke* service from the OPC Foundation. This is mainly because for these services an OPC UA server has to keep some state. Nevertheless, this kind of state does not conflict with the stateless requirement of the REST architecture. For example, the upload/deletion of a web-page, based on some kind of REST API, also creates some new state on the server. Our approach to solve this problem is to map subscriptions into the information model of an OPC UA server. A similar idea was already introduced by the OPC Foundation in Part 14 for the PubSub configuration. However, one problem still remains. Each *Node* in an OPC UA server must have a unique *NodeId*. If *Nodes* are generated dynamically during runtime, it has to be ensured, that previously assigned *NodeIds* are not used again for another *Node*. However, for session-less clients the *NodeId* must also be unique across server crashes/restarts, because a session-less client might not take notice of such an event. A possible solution for this problem is the introduction of a so-called *RuntimeNamespace*. All dynamically created *NodeIds* will be assigned to this special *Namespace*. The *Namespace*-URI will be automatically generated and has to be unique. If the *Namespace* is full, or for example, a restart occurs, all dynamically generated *Nodes* can be safely deleted by simply generating a new URI for the *RuntimeNamespace*. This approach will ensure that each *NodeId* for a dynamically generated *Node* is unique across server crashes/restarts.

F. Conclusion

In this Section, we discussed our approach. Some of our concepts were accepted by the OPC Foundation and now are part of the newest release of the OPC UA specification. However, we also discussed the differences between both concepts in the mapping to the HTTP protocol. While our approach uses different HTTP methods and therefore is able to utilize the semantics behind these methods (idempotence, safety, failure handling, ...), the mapping of the OPC Foundation only uses the POST method. Additional concepts, like the realization of subscriptions, can be easily applied to the standardized proposal too. Nevertheless, the most important feature, browser support, is not really possible with the currently released version of the OPC UA specification, because of the HTTP mapping. In the end, our proposal would be able to add some new useful features to OPC UA and could also be used to reduce the complexity of OPC UA for web-developers.

V. Prototype

In this Section, we will give an overview of our prototype and also show some of the implemented features.

A. Overview

Our prototype is based on the Java OPC UA stack implementation of the OPC Foundation. The following features were implemented:

- *Read* and *Write* service (including batch support)
- *Browse* and *BrowseNext* service
- *TranslateBrowsePathsToNodeIds* service
- *Call* service
- *SessionlessInvoke* Base (*NamespaceUris*)
- *SessionlessInvoke* Optimized (*UrisVersion*)
- (+) MIME-Type handling
- (+) Content negotiation
- (+) ResolvePath (Redirections)
- (+) ExternalReferences

Most of the services are well-known to the OPC UA community, but it should not be a surprise that the access pattern is sometimes quite different to a standard OPC UA server. For example, to collect all necessary information for a *NodeRepresentation*, more than one OPC UA service must be invoked. The services which are marked with a (+) will now be explained in greater detail. Under the term **MIME-Type handling** we address the feature, that arbitrary files can be served with the correct MIME-Type, based on the *MIME-Type-DataType-Property* (see also Section IV-D). **Content negotiation** is the standard way in the web for a client to request a certain representation of a resource. To be more concrete, a client is able to specify the encoding, for example, ”application/opcua+uajson” or ”application/opcua+uabinary”. **ResolvePath** was introduced to offer a more comfortable way for the OPC UA concept programming against the *TypeDefinition*. Just consider the following URL ”<host>/Objects/0:Server/NamespaceArray”. Resolving the URL would lead to a redirect to the URL ”<host>/1/i=2255”, based on the first entry of the *TranslateBrowsePathsToNodeIds* service response. However, not every *Node* can be addressed by this concept, because the *BrowsePath* is not unique (see also OPC UA Part 3 for further details). **ExternalReferences** can be derived from *ExpandedNodeIds*, if certain additional restrictions hold. For example, a server-URI has to be a valid URL to the REST endpoint of the OPC UA server.

B. Features

As already mentioned in Section II, for a RESTful OPC UA server it is necessary to define resources and their representations. In IV-B several representations were introduced. Because of space limitations, we will only focus on the ”application/opcua.NodeRepresentation+json” representation (see Listing 2). The basic structure of this representation consists of three parts: **attributes**, **references** and **forms**. In the attributes section all *Attributes* of a given *Node* are summarized, while the reference section offers all *References* of the *Node*. However, the form section might not be expected by an OPC UA

user. One of the preconditions of a RESTful API is, that a client should be able to explore all service functions without additional documentation [11]. If one remembers the basic concept behind REST (Section II-A), we not only have to display somehow that there are additional resources, for example, the "StaticVariableFolder" resource, instead, we also have to provide the client with the knowledge how to access them. This is done by introducing a new field in the OPC UA *NodeId* structure with the name "href". The value of the "href" field is a valid URL to the target *Node* of the *ReferenceDescription* structure. The href field "<host>/2/s=1:Boolean" references the string *NodeId* "Boolean" based on *NamespaceIndex* "1" and the *SessionlessInvoke urisVersion* "2". Notice, that this additional field does not force OPC UA architects to add additional information to already existing OPC UA information models, instead, it can be easily generated automatically out of existing information. But expressing all possibilities with links would add a large amount of data to each representation. For that reason, the form section is introduced. In this section, it is possible to express standard functions of the service based on URI templates [13]. In the forms section of Listing 2 an example for a possible *Read* URI template is given. For performance reasons, this section can also be suppressed through special filter settings.

```
{
   "attributes" : {
    "NodeIdValue": {
      "NodeId": {
        "namespace": 1,[...]
        "href": "<host>/1/s=1:Boolean"
      },[...]
    },[...]
    "Value": {
      "href": "<host>/1/s=1:Boolean/Value",[...]
    }
  },
  "referencesStatusCode": 0,
  "references" : {
    "(!i=0:47)s=1:StaticVariablesFolder": {
      "nodeId": {[...]
        "href": "<host>/1/s=1:StaticVariablesFolder"
      },[...]},[...]
  },
  "forms": {
    "DefaultRead": {[...]
      "href": "<host>/1/s=1:Boolean/{attributeName}
               {?opcuaAuthenticationToken}",
      "method": "GET",
      "uriTemplate": true,
      "jsonschema": {
        "type": "object",
        "properties": {
          "attributeName": {
            "type": "string",
            "description": "The attribute name",
            "enum": ["NodeId",...]
          },[...]
        },
        "required": [ "attributeName" ]
      }
    },[...]
  }
}
```

Listing 2. Example *NodeRepresentation* (simplified)

A typical graphical OPC UA Client, like UAExpert [22], often offers some kind of template if the OPC UA *Call* service is invoked by the user. This template is based on well-known OPC UA properties, which are part of each *Method-Node*. However, a typical web-client is not aware of these OPC UA specific conventions. Because of that, the form section also provides some additional information if OPC UA *Method-Nodes* are involved. The most important part of such a representation is depicted in Listing 3. The representation provides all necessary information, including the target URL, the HTTP method, and the expected Content-Type. The section requestSchema and responseSchema provides a URL to the corresponding schema description for the request/response Content-Type. Our Prototype uses JSON schema [23] for this purpose, which is well-known in the web and also offers client-side validation. However, based on HTTP content negotiation it is also possible to serve other schema descriptions.

```
{
  "href": "<host>/1/s=1:AttributeServiceSet",
  "method": "POST",
  [...],
  "accepts": ["application/opcua.CallRequest+json"],
  "requestSchema": {
    "href": "<host>/1/s=1:Read/RequestSchema"
  },
  "responseSchema": {
    "href": "<host>/1/s=1:Read/ResponseSchema"
  }
}
```

Listing 3. Representation of *Methods* in the form section.

The next example focuses on the pagination of OPC UA. A lot of services in OPC UA may return a large amount of data and because of that, many **Next* services were introduced by the OPC Foundation. On the web, the typical pagination paradigm is to offer some URL, which returns the next results (HATEOAS). Our prototype also supports this kind of paradigm. If not all results are part of the response, an additional continuationPoint field is present (see also Listing 4). If a client sends a GET request to the specified URL, the next part of the result is returned. A nice property of this paradigm is the fact, that we do not have to introduce multiple **Next* services (e.g., *BrowseNext*, *QueryNext*, ...), instead we only have to add this field to the corresponding representation.

```
"continuationPoint": {
  "continuationPointId": "AQed",
  "href": "<host>/1/i=85?continuationPoint=AQed"
}
```

Listing 4. **Next* services powered by HATEOAS.

VI. Related Work

Below, we discuss related work, which also proposes some kind of RESTful OPC UA interface.

RESTful Industrial Communication [5]: The authors also introduced a RESTful OPC UA architecture. However, because the authors have not shown any solution for the dynamic *Namespace-* and *ServerArray* problem, we think that some additional restrictions must be placed on a server which uses this concept (e.g., the *Namespace-* and *ServerArray* cannot be changed). In contrast, our concept can be used by any OPC UA server, without introducing additional restrictions. Besides the points mentioned above, also one feature for a RESTful architecture, HATEOAS, is not part of the concept.

OPC UA over CoAP [6]: This draft is also working on an OPC UA REST interface. However, the approach seems to build on the findings of [5], which were already discussed above. Because of that, the same restrictions should apply.

Industrial Middleware [7]: This paper discusses a linked data architecture for OPC UA, also leveraging an OPC UA REST API. The concept also uses the HATEOAS concept for interconnecting different *Nodes*. In the end, the authors seem to use a similar concept as the authors of [5], for introducing statelessness in OPC UA, and therefore the same restrictions should apply.

Protocol interoperability of OPC UA [8]: Also, this proposal is inspired by [5] and therefore the same restrictions should apply. However, this concept only maps 7 services from OPC UA to a REST API, while our concept even has a proposal for subscriptions. In addition, we are also not sure if a certain criterion for a REST API is covered, HATEOAS.

HyperUA [24]: HyperUA offers a very nice web-based interface. The problem of OPC UA, that each service which uses the *Namespace-* and *ServerArray* needs an active session, is solved by still creating sessions and encode all necessary information into the URLs. While this approach also includes the HATEOAS paradigm and therefore offers a RESTful feeling, finally, each client still has to create a session and therefore the service cannot fully leverage all benefits of REST.

dataFEED OPC Suite [25]: The dataFEED OPC Suite introduces a so-called REST client API. After taking a closer look at the documentation and at the evaluation version of the software framework, we identified that the REST API is some kind of data push API. Basically one can define so-called "Actions", which will be invoked by user-defined conditions and after that send a message to a user-defined REST endpoint. In the end, we came to the conclusion, that the dataFEED OPC Suite REST API has other goals than our API.

KEPServerEX [26]: Kepware published an IoT-Gateway plugin for their OPC UA software framework KEPServerEX. This plugin also offers a REST interface to access OPC UA data. The basic concept behind this REST API is three predefined URLs, which allow someone to execute some kind of read, write and browse service. However, these services only have the name in common with the corresponding OPC UA services and therefore are completely disjoint with our approach. For example, the browse service returns all "tags", which are configured for the given REST server interface and not the *References* of OPC UA *Nodes*.

VII. Summary and Outlook

In this paper we analyzed why OPC UA cannot be considered stateless, even for trivial services like *Read*, *Write* and *Browse*. After having identified the challenges, which prevent us from introducing stateless requests into OPC UA, we discussed an efficient way how this problem could be solved and contributed parts of our concept to the newest release of the OPC UA specification (V1.04). In addition, we presented our approach how the REST architecture can be mapped to OPC UA and also gave some insights into our design decisions and goals. Last but not least, we presented our RESTful OPC UA prototype, to show the feasibility of our concept and several benefits of our architecture, as for example the ability so serve PDFs to a standard web browser.

In Section IV-E we outlined our concept for session-less subscriptions. However, because our SDK did not offer an implementation for subscriptions, we were not able to include the concept in our prototype. Because of that, we switched to the C++ SDK of Unified Automation and started to implement the missing feature.

References

[1] "Reference architectural model industrie 4.0," https://opcconnect.opcfoundation.org/2015/06/opc-ua-in-the-reference-architecture-model-rami-4-0/, 2018.

[2] Y. Li, S. Hu, W. Tao, and B. Li, "Research on the industrial network architecture of northbound interface," in *Chinese Automation Congress*, 2017.

[3] "Iec 62541: Opc unified architecture," Standard, 2010.

[4] R. T. Fielding, "Architectural styles and the design of network-based software architectures," Ph.D. dissertation, 2000.

[5] S. Gruener, J. Pfrommer, and F. Palm, "Restful industrial communication with opc ua," *IEEE Transactions on Industrial Informatics*, vol. 12, no. 5, 2016.

[6] P. Wang, C. Pu, H. Wang, J. Wu, Y. Yang, L. Shao, and J. Hou, "OPC UA Message Transmission Method over CoAP," IETF, Internet-Draft draft-wang-core-opcua-transmission-03, 2018, work in Progress.

[7] M. Graube, L. Urbas, and J. Hladik, "Integrating industrial middleware in linked data collaboration networks," in *IEEE Emerging Technologies and Factory Automation*, 2016.

[8] H. Derhamy, J. Rnnholm, J. Delsing, J. Eliasson, and J. van Deventer, "Protocol interoperability of opc ua in service oriented architectures," in *IEEE Industrial Informatics*, 2017.

[9] G. R. Andrews, "Paradigms for process interaction in distributed programs," *ACM Computing Surveys*, vol. 23, no. 1, 1991.

[10] D. Garlan and M. Shaw, "An introduction to software architecture," Tech. Rep., 1994.

[11] R. T. Fielding, "Rest apis must be hypertext-driven," http://roy.gbiv.com/untangled/2008/rest-apis-must-be-hypertext-driven, 2008.

[12] J. Miranda, J. Cabral, S. Banerjee, D. Grossmann, C. F. Pedersen, and S. R. Wagner, "Analysis of opc unified architecture for healthcare applications," in *IEEE Emerging Technologies and Factory Automation*, 2017.

[13] R. T. Fielding, M. Nottingham, D. Orchard, J. Gregorio, and M. Hadley, "Uri template," RFC 6570, 2012.

[14] M. Nottingham, "Uri design and ownership," RFC 7320, 2014.

[15] "Github replaces rest with graphql," https://githubengineering.com/the-github-graphql-api/, 2018.

[16] "Amazon s3," https://docs.aws.amazon.com/AmazonS3/latest/API/Welcome.html, 2018.

[17] "Google gmail," https://developers.google.com/gmail/api/v1/reference/, 2018.

[18] "Apple news api," https://developer.apple.com/library/content/documentation/General/Conceptual/News_API_Ref/index.html#//apple_ref/doc/uid/TP40015409-CH2-SW1, 2018.

[19] "Twitter api," https://developer.twitter.com/en/docs/api-reference-index, 2018.

[20] "Rest api multiple-request chaining," https://github.com/mikestowe/REST-API-Multiple-Request-Chaining, 2016.

[21] "Dash web application," https://www.thedash.com/, 2018.

[22] "Unified automation: Uaexpert," https://www.unified-automation.com/de/produkte/entwicklerwerkzeuge/uaexpert.html, 2018.

[23] A. Wright and H. Andrews, "JSON Schema: A Media Type for Describing JSON Documents," IETF, Internet-Draft draft-handrews-json-schema-00, 2017, work in Progress.

[24] "Projexsys: Hyperua," http://projexsys.com/hyperua/, 2018.

[25] "Softing: datafeed opc suite," https://industrial.softing.com/en/products/software-connectivity/opc-suite-servers-and-middleware.html, 2018.

[26] "Kepware: Kepserverex," https://www.kepware.com/en-us/products/kepserverex/, 2018.