Integration of OPC Unified Architecture with IIoT Communication Protocols in an Arrowhead Translator

Jesper Rönnholm

Engineering Physics and Electrical Engineering, master's level
2018

Luleå University of Technology
Department of Computer Science, Electrical and Space Engineering
Integration of OPC Unified Architecture with IIoT Communication Protocols in an Arrowhead Translator

Jesper Rönnholm

Master’s thesis

Dept. of Computer Science, Electrical and Space Engineering
Luleå University of Technology

Supervisors: Jerker Delsing, Hasan Derhamy

2018
Abstract

This thesis details the design of a protocol translator between the industrial-automation protocol OPC UA, and HTTP. The design is based on the architecture of the protocol translator of the Arrowhead framework, and is interoperable with all of its associated protocols. The design requirements are defined to comply with a service-oriented architecture (SOA) and RESTful interaction through HTTP, with minimal requirement of the consuming client to be familiar with OPC UA semantics. Effort is put into making translation as transparent as possible, but limits the scope of this work to exclude a complete semantic translation. The solution presented in this thesis satisfies structural- and foundational interoperability, and bridges interaction to be independent of OPC UA services. The resulting translator is capable of accessing the content of any OPC UA server with simple HTTP-requests, where addressing is oriented around OPC UA nodes.
Preface

I was first introduced to the opportunity of doing this project as part of master’s thesis in 2015 by Prof. Jerker Delsing. He was the coordinator of the Arrowhead Project, which at the time was the largest industrial automation project in Europe, with 78 partners and a budget of 68 million euro. Arrowhead was dedicated to overcoming many of the challenges anticipated for future industrial automation. By the time I became involved, it had already produced a framework necessary for interoperability within modern, large-scale systems. Many of Arrowhead’s subordinate pilot projects were also nearing completion, including the Arrowhead protocol translator, designed by Hasan Derhamy, who later became my supervisor for this thesis project.

To be able to understand the "internet" part of "internet of things", I would spend the rest of 2015 taking some courses in networking, and trying to understand what Arrowhead was all about - something that felt very abstract to me at the time. It was not until mid-spring in 2016 that my work on this thesis project would actually begin, and I would start to laboriously form an understanding of OPC UA, which was the protocol suite I was to design a translation scheme for. Because of the sheer size of OPC UA’s specification and its protocol stack, it would take a long time before any real progress was made.

In its entirety, my work on this thesis was spread over two longer periods; the first in 2016, running several months into the autumn to produce a rudimentary proof-of-concept. The second was in 2017, when I returned to (under Hasan’s guidance) compile the work into a useful translation scheme. This process culminated around the beginning of summer in 2017, at which point I had been allowed to help put the results of the work into a conference paper presented at INDIN 2017.

It would not be until autumn in 2017 that all the work, its material and loose ends would be wrapped up. At that point, well over a year had passed since I first began working on OPC UA.

I want to thank Hasan and Jerker for all their patience, guidance and support. In addition to their supervision of my thesis project, they widened my perspective and gave me insight into their research in the Arrowhead project. For all this, I am very grateful.

Luleå, May 2018
Jesper Rönnholm
## Contents

1 Introduction  
   1.1 Industrial Internet of Things  
      1.1.1 OPC UA’s role in ICoT  
   1.2 Thesis Motivation  
      1.2.1 Thesis Contribution  
   1.3 Terminology  

2 Related Work  
   2.1 Themes  
      2.1.1 Direct Integration  
      2.1.2 Hybrid Middleware with Mapping to Other Standards  
      2.1.3 Gateway Middleware  
      2.1.4 Adaptations  
      2.1.5 Revisions & Development of the OPC UA Standard  

3 Body  
   3.1 Design Goals  
      3.1.1 The Arrowhead Protocol Translator  
      3.1.2 Service-Oriented Architectures (SOA)  
      3.1.3 Interoperability Layers  
      3.1.4 Translation Transparency  
   3.2 OPC Unified Architecture  
      3.2.1 Nodes & References  
      3.2.2 Nodeclasses  
      3.2.3 Attributes  
      3.2.4 OPC UA Data Types  
      3.2.5 The Address Space  
      3.2.6 OPC UA Services  
      3.2.7 Service Interaction  
      3.2.8 The OPC UA Communication Stack  
      3.2.9 Summary  
   3.3 REST  
      3.3.1 Key Features  
      3.3.2 Uniform Interface Concepts  
      3.3.3 REST in a Service-Oriented Context  
      3.3.4 HyperText Transfer Protocol (HTTP)  
   3.4 Translation Solution Outline  
      3.4.1 OPC UA Stack Review  
      3.4.2 Translation Interface  
      3.4.3 Translation Service Mapping Outline  
      3.4.4 State-dependence of OPC UA Services
4 Design

4.1 Summary of Design Goals ........................................... 35
4.2 Addressing Scheme .................................................. 35
4.3 Semantic Dependency (and what to do about it) .......................... 35
   4.3.1 Semantic Dependency between Interoperability Layers ............. 36
   4.3.2 Complexity of Interaction Patterns .............................. 36
   4.3.3 In this work ................................................... 36
4.4 Limitations Imposed on Design ....................................... 37
   4.4.1 NodeID Formats ............................................... 37
4.5 Service Mapping ...................................................... 37
   4.5.1 Translation approach to OPC UA service differentiation .......... 37
   4.5.2 Service Translation Alignment .................................. 37
4.6 Service Differentiation within Methods ................................ 38
   4.6.1 GET Requests ................................................... 38
   4.6.2 PUT & POST Requests ......................................... 40
   4.6.3 DELETE-requests ............................................... 43
4.7 Service Mapping Summary ............................................ 43
4.8 Type Support .......................................................... 44
   4.8.1 Plain-Text Payloads ............................................ 44
4.9 Request Formats ........................................................ 45
   4.9.1 Addressing Scheme ............................................. 45
   4.9.2 Payload Formats ................................................ 45
4.10 Response Formats ..................................................... 47
   4.10.1 Contents of Response Data ..................................... 47

5 Implementation .......................................................... 49

5.1 Arrowhead Translator ................................................. 49
   5.1.1 Translation Interface .......................................... 49
   5.1.2 Intermediary Format Interface .................................. 50
5.2 OPC UA (client) Translator Spoke .................................... 50
   5.2.1 Architecture .................................................... 50
5.3 Test Environment ...................................................... 52
   5.3.1 Node-OPC UA Server ............................................ 52
   5.3.2 Unified Automation Demo Server ................................ 52
   5.3.3 Open62541 ...................................................... 52
   5.3.4 3rd-party OPC UA Client Software ............................... 52
   5.3.5 HTTP Tools ..................................................... 53
   5.3.6 CoAP Wheel Loader Testbed .................................... 53

6 Evaluation ............................................................... 54

6.1 Constraints imposed by translation ................................... 54
   6.1.1 Services ......................................................... 54
   6.1.2 Data Types ..................................................... 55
   6.1.3 Security ......................................................... 55

7 Conclusion ............................................................... 57

8 Future Work ............................................................ 58

8.1 Expanding Translation Service Coverage .............................. 58
8.2 Semantic Translation .................................................. 58
   8.2.1 Parametrization ................................................. 58
   8.2.2 Self-descriptive Interfaces ..................................... 58
   8.2.3 Parsing of Response Payloads .................................. 59
8.3 Reverse Mappings & Discovery ........................................ 59
8.4 gRPC ................................................................. 59
List of Figures

3.1 Block diagram of the base translator architecture .................................. 14
3.2 In the semantic perspective, references represent a relationship with implicit
    semantics in between two nodes, but are in practice stored as list entries on
    their source nodes. ........................................................................ 17
3.3 The OPC UA node classes and their respective attributes. ............................. 19
3.4 Base of the ReferenceType tree. Every reference in an address space are
derived from these. ........................................................................ 22
3.5 The services defined by OPC UA. ......................................................... 23
3.6 Interaction diagram of secure channel setup, session creation and session activa-
tion. The principal service calls are marked by the grey area in the diagram. They
make up the actual data exchange, rather than communication management. [33] 24
3.7 The communication stack as illustrated by the OPC UA specification [16]. 25
3.8 URL format for locating resources via HTTP. ............................................. 28
3.9 One possible visualization on how OPC UA interoperability barriers can be
divided. ......................................................................................... 31
3.10 The services defined by OPC UA. Services marked by * are considered state-
less [23]. ...................................................................................... 34
3.11 The services with inherent support for stateless, RESTful translation ... 34
4.1 NodeID represented as a URI path .............................................................. 37
4.2 NodeID URL structure for an arbitrary node. ............................................. 41
4.3 NodeID URL structure for an arbitrary attribute of a node. ......................... 41
4.4 Generalized URL format for accessing a server application resource, node or
attribute. ...................................................................................... 45
4.5 URL/Payload parameter format for adding a node with a PUT-request. ....... 46
4.6 URL/Payload parameter format for adding a reference with a POST-request. 46
4.7 URL/Payload parameter format for deleting a reference with a POST-request. 47
4.8 URL/Payload for writing attributes with a POST-request. ......................... 47
5.1 Arrowhead Translator request payload format (XML) for setting up a pro-
tocol spoke-pair that enables transparent translation ................................ 49
5.2 Response format for translation spoke setup. Translation endpoint ID, IP
and port. The IP and port are used as the URL-authority in the OPC
UA-oriented HTTP requests. ................................................................ 50
List of Tables

3.1 Principles of Service-Oriented Architectures ........................................... 15
3.2 OSI layers vs interoperability layers vs solutions ..................................... 16
3.3 The list of all possible attributes the various node classes may contain, listed with their corresponding (fixed) attribute indexes ......................................... 20
3.4 Built-in DataTypes as defined by the OPC UA specification [16] ................. 21
3.5 OPC UA stack compared to other protocols and interoperability layers ........ 26
3.6 The semantic meaning of methods ............................................................. 29
3.7 URI semantics in combination with methods .............................................. 29
4.1 The Service Mapping of OPC UA Services to the RESTful HTTP Intermediary Format .......................................................... 43
Chapter 1

Introduction

1.1 Industrial Internet of Things

With an ever expanding fauna of interconnected intelligent devices in Industrial Automation, one of the key paradigm shifts behind the Industrial Internet of Things (IIoT) has been the enabling of direct internet access down to the sensor-level in the ISA-95 automation pyramid [32]. Following the advent of Industry 4.0, the traditional hierarchical layout of the automation pyramid is expected to be replaced by an open, distributed layout [4] composed of local automation clouds [8] that are more open and accessible than their previous counterparts. As the dependence on traditional hardware- and software intermediaries for networking intelligent devices is eliminated, and intelligent ecosystems are no longer limited by them in scale, or confined to hierarchical layout [4], it is expected that automation systems will grow very large and involve a large number of separate stakeholders. For all of these to integrate without dramatically increasing the engineering cost of each system, a new problem to take center stage is the issue of interoperability [7, 9, 8]. A diversity in communication protocols, information semantics, and data encoding are some of the examples presented by the Arrowhead project [7] as interoperability challenges that future automation systems must address.

1.1.1 OPC UA’s role in IIoT

Since the release of its specification in 2010, OPC UA has gained a steadily increasing support in the community, and seen a number of adoptions in novel solutions that seek to integrate with one or more of its promising features. When approaching the industrial internet of things, OPC UA has sought to provide a wide foundation for accessing, building and managing resources in a service-oriented approach. This includes a client-server architecture that defines a standardized interface to access resources that collectively form a graph-based information model.

OPC UA is, unlike its predecessor (Classic OPC), platform independent. It also allows for its information model to be extended while also being fully self-descriptive. Therefore, all systems that implement the fundamental OPC UA nodeset can share a common understanding for any body of information based upon it. However, the tradeoff that OPC UA make for its powerful foundation is that, to access a resource hosted on an OPC UA platform, all third-party agents are required to comply with the OPC UA communication stack (which essentially encompass the entire vocabulary of OPC UA).

Hence, while OPC UA boast itself platform-independent, it is at the same time exclusive to its own interface. From a standpoint of integration, this demand of compliance impedes on its interoperability potential with other systems as it effectively creates a barrier that any implementation must overcome. Since the release of OPC UA, various efforts have been made to circumvent this problem by introducing an intermediary to bridge the gap between compliant and non-compliant systems. In most cases this has meant either setting up a gateway to formalize client interaction with the server through a web API, or incorporating interoperability solutions into a centralized gateway server.
1.2 Thesis Motivation

For OPC UA to fully realize its potential as a component in the Industrial Internet of Things, it must be able to easily integrate with the IoT-ecosystem. The amount of ongoing research has highlighted the high demand for- and possible steps towards making this a reality [22, 29, 23, 28]. Arrowhead is no exception, and the benefits of incorporating OPC UA into its list of supported protocols has long been appreciated. Being able to interact with other IoT protocols such as HTTP, CoAP and MQTT is part of Arrowheads vision in opening OPC UA up to the RESTful web, and central to the issue of interoperability.

1.2.1 Thesis Contribution

This thesis discuss solutions for enabling greater interoperability with OPC UA (OPC Unified Architecture), and presents a design for client-side OPC UA to REST-translation by a mapping to HTTP that impose a minimum of semantic dependency. It specifies how OPC UA may integrate with Arrowhead’s transparent protocol translator, and future challenges for complete interoperability with third-party systems.
1.3 Terminology

Important terms introduced- or frequently used in this thesis:

**Semantic** The meaning of a data field, or the value of a data field.

**Service-Oriented Architecture** A set of design principles related to interoperability.

**Interoperability Layer** A categorization on interoperability, being either structural, foundational or semantic. Describes the interoperability goals for a given aspect of communication.

**Semantic Dependency** Leakage of endpoint semantics into the consumer interface of a given interoperability solution - introducing the need for prerequisite understanding in order for the solution to work. This fundamentally inhibits *loose coupling*.

**Transparent Translation** A translation scheme that is invisible to the client, and that does not require additional pre-configuration or understanding of the endpoint.

1 See section 3.1.2 for reference.
Chapter 2

Related Work

2.1 Themes

There is a wide array of different directions in which previous and ongoing interoperability efforts have moved. Most of the aspects of OPC UA have been scrutinized, and the resulting solutions vary both in scope and approach. A common theme is the deployment of gateways to provide a web-based API [30, 5], or to use protocol adapters [27, 26, 6] in conjunction with an aggregating server - either of which can be summarized as dedicated middleware. Another common approach has been to propose adaptations to various parts of the OPC UA communication stack [22, 29, 23, 37, 36] with the intention of making it more friendly to integration in general. Moreover, active revision of the OPC UA specification [18] is still ongoing, with additions having been released very recently [13] by the OPC Foundation.

2.1.1 Direct Integration

With OPC UA having a solid industrial support, there is a clear incentive to use it as a core technology. This has motivated integration of complementary, third-party technologies with various aspects of OPC UA. One example of this is [28], which presents an approach to integrating the 4DIAC framework with OPC UA. 4DIAC (Framework for Distributed Industrial Automation and Control) is a reference architecture, based on the IEC-61499 standard, intended to enable a flexible, model-based development of distributed control systems. A key component in IEC-61499 is the Function Block (FB), which serves to encapsulate control functionality in a way that is extensible and easy to interface with other technologies. The approach of [28] has already seen support in domain-specific applications. A demonstration of this is presented in [20] to help with classic concepts such as complexity management and vertical integration. Combined implementation of IEC-61499 and OPC UA is also further discussed in [21].

Another example of interoperability related to OPC UA can be seen in [6], where data from a Control Area Network (CAN) leverage the information modeling in OPC UA to provide integration of CAN-devices with the IIoT infrastructure.

2.1.2 Hybrid Middleware with Mapping to Other Standards

OPC UA is not alone in the domain of service-oriented standards. There are multiple technologies with aspects that run in parallel to the functionality of OPC UA. Addressing the topic of interoperability on a more general level, some works have therefore looked at ways of interfacing OPC UA with other protocols, or creating a unified interface.

This has been the case with Devices Profile for Web Services (DPWS), which like OPC UA has been regarded as enabler for SOA at device-level in the automation pyramid. Device-level SOA is a key concept in vertical integration. DPWS is designed to enable web messaging on resource-constrained devices, and features device-specific services as well as a set of built-in services that enable standardized discovery, metadata access and publish/subscribe. These are all features that are present in OPC UA as well, and there has been work done to evaluate [3, 2] how OPC UA and DPWS may complement each other, and
work in synergy to provide further benefits when converging to a common platform. In [25], an implementation solution is proposed to accomplish a unified event management that enable the two standards to interoperate. Going even further to reach a complementary solution, [31] proposes a middleware platform that combine parts of OPC UA and DPWS.

Another paper to discuss hybridization is [29], which provides an analysis on the differences between OPC UA and the communication protocol DDS (Data Distribution Service). Specifically, it provides a mapping of OPC UA data types to DDS, a discussion on how the protocols are architecturally different, and what a hybrid implementation might look like.

### 2.1.3 Gateway Middleware

Rather than striving towards hybridization, some interoperability solutions have instead opted for translation. In the case of OPC UA, a common approach has been to enable web-based access to OPC UA through HTTP. Although this was already supported to some extent in the original specification releases until the deprecation of HTTP/SOAP, proprietary solutions like HyperUA [30] were developed to accommodate such a protocol bridging.

**HyperUA**

HyperUA [30] was released in April 2013 by Projexsys, and marketed as enabling OPC + Web with the intent of adding RESTful web capabilities to OPC UA - something that the native OPC UA specifications was, and still is, lacking. In its presentation, it points out that the SOAP/XML over HTTP transport option (which was later deprecated along with WS-SecureConversation in release 1.03 [12]) in the OPC UA communications stack isn’t RESTful, and therefore not very useful for web connectivity. Instead of SOAP, the HyperUA gateway-server would expose its content through an HTML API, which encapsulated OPC UA client functionality, and offered web-based access to OPC UA. As it featured an HTML-based mapping to OPC UA functionality, it also enabled simple browser access, where some OPC UA services could be composed and called through HTML forms.

**Kepware**

Another proprietary solution that provide some interoperability with OPC UA is Kepware’s KEPServerEX 6 [26], which expose its data as legacy OPC data (tags). In Classic OPC, tags represent addressable I/O-points and can be seen as OPC’s equivalent to OPC UA nodes. The OPC UA interface is therefore enabled by exposing these tags as nodes in OPC UA address spaces that can be accessed by OPC UA clients.

Interoperability can then be extended by a gateway [27] to allow third-party access to tags through REST- and MQTT-client APIs.

**Cavalieri et al.**

A paper presented by an Italian group [5] explored a design for a web-based platform to allow for access to OPC UA servers. Much like HyperUA, this solution was oriented as middleware, but with some effort to provide abstraction on top of OPC UA semantics. The authors acknowledged the need for adapting interoperability solutions for third-party users lacking knowledge of OPC UA concepts. To address this challenge, they formulated a number of substitute concepts that would aggregate some of the OPC UA functionality to enable a more generalized interface. Four services was introduced as interaction with the gateway: SecureAccess, GetDataSources, ReadInfo and Monitor. These would then be used as abstraction for authentication with the gateway, retrieve OPC UA server lists, retrieve node sets (from address space hierarchies) and monitor value changes in variable-type nodes.
The semantic translation of this solution is subscription-oriented, and map 5 of the 22 possible OPC UA node-attributes to custom representations in the gateway, with an ad-hoc attribute called Monitorable introduced to indicate if a node can be monitored through the gateway.

Drawbacks of Gateway Middleware

The primary problem with interoperability solutions such as gateways and protocol adapters is that they impose application logic that requires custom configuration per site, which doesn’t scale well with each additional protocol added to the mix. As the complexity of a centralized gateway grows, the solution will become more brittle, and resists change.

2.1.4 Adaptations

Making OPC UA more RESTful

Two papers submitted by Grüner et al. [22, 23] investigated how RESTful architectures could improve on OPC UA performance, and industrial settings in general, by allowing them to operate more like RESTful web services. The adaptations in question would feature stateless service requests, caching layers and reduced communication overhead. Combined with UDP-based transport, an adaptation to the OPC UA binary protocol would allow for almost an order of magnitude-improvement in throughput for short-lived interactions. The papers further discussed the motivation and benefits of RESTful OPC UA, and how it would be better adapted to IoT applications involving resource-constrained devices and/or load-balancing in large networks.

CoAP-transport for OPC UA

Despite the HTTP/SOAP stack transport option of OPC UA being deprecated since 2015 the use of HTTP purely as transport was still appreciated as a technical solution, and inspired an IETF design draft [37, 36] that proposed the development of CoAP to be used as a transport alternative. This would involve using UDP as a foundation, and employing DTLS (Datagram Transport-Layer Security) as a security option. The use of CoAP would not affect OPC UA semantics, however. Implementing CoAP would only have implications on foundational and structural interoperability, and not affect OPC UA service interfaces.

When looking towards semantic contexts, this draft also analyzed how well OPC UA conforms to the REST architecture. An interesting point was made on how CoAP was uniquely suited (in contrast with HTTP) to support publish/subscribe patterns.

2.1.5 Revisions & Development of the OPC UA Standard

OPC UA is still seeing active revision of its specification [12] and additions to the standard [17] are also in development.

Deprecation of SOAP over HTTP

As implied above, a well-known example of revision to OPC UA was the OPC Foundation’s 2015 deprecation of the SOAP/HTTP transport option in the OPC UA stack. This change was motivated by the lack of industrial approbation of WS-SecureConversation, which would’ve undermined SOAP/HTTP as a secure mode of transportation. Hence, to implement HTTP-based transport, HTTPS is the only remaining alternative for security.

---

1It will be explained in later chapters how OPC UA’s own HTTP-based stack transport does not satisfy RESTful, semantically decoupled interaction.
OPC UA Pub/Sub

PubSub \cite{17} is a new OPC UA communication pattern that operates, in contrast to OPC UA’s conventional client-server layout, between publisher and subscriber.

In the standard OPC UA client-server layout, sessions are used to maintain communication between a single client and server. Client-server subscriptions are confined to a single session, and therefore not shared between clients. This has been a major impedance on interoperability, which led the OPC Foundation to develop a second communication model called PubSub.

In PubSub, requests and responses are never exchanged directly, and no communication context is maintained. There, applications act as publishers and/or subscribers without any knowledge of other publishers/subscribers. All communication is between application and Message Oriented Middleware (MOM), that act as either broker-less\footnote{Broker-less Message Oriented Middleware would operate over UDP multicast.} or broker-based\footnote{Broker-based Message Oriented Middleware would operate over standard messaging protocols like AMQP or MQTT.} form to forward messages.
Chapter 3

Body

3.1 Design Goals

The ultimate goal of this work is to present a protocol translation scheme that provides the highest level of interoperability on top of all current solutions.

One of the most central topics on interoperability design in general is Service-Oriented Architectures - a set of principles meant to increase the interoperability of independent systems by standardizing software interfaces and increasing implementation flexibility.

The parent project of this thesis, Arrowhead, is centered on creating a service-oriented framework, and is so the foundation for this works design. More specifically, the protocol translator presented by Derhamy in [10] serve as the outset for this work - which will be a derivative of its design requirements.

3.1.1 The Arrowhead Protocol Translator

The Arrowhead Translator [10] is designed with a hub-spoke layout, where each spoke represent an IoT communication protocol. When a request across two different protocols is made, the hub injects a translation (which is transparent to the requesting client) that creates a pipelined protocol bridge.

The protocol bridge works by performing two translations. The first translation, from any protocol, is made into an intermediary format designed to retain as much of the source protocol request context as possible. Once there is an intermediary protocol representation in place, the target translation can proceed. The implementation of an intermediary translation for all protocols enables them to only require one translation scheme. Hence, for a protocol to be able to interact through the hub, it only requires a mapping to the intermediary representation.

To this end, the OPC UA translation must satisfy the design of the Arrowhead Translator [10] by a mapping to its intermediary format. Furthermore, the OPC UA translator implementation must deploy as a protocol spoke that can serve this mapping.

Figure 3.1: Block diagram of the base translator architecture
3.1.2 Service-Oriented Architectures (SOA)

There is a multitude of design principles that would be desirable for any interoperability solution. As SOA (Service-Oriented Architecture) is already an established design paradigm for Arrowhead, it is included here as well:

<table>
<thead>
<tr>
<th>Principle</th>
<th>Description</th>
<th>Application in translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>Standardised service contract</td>
<td>Used for services to express their capabilities and method of interaction.</td>
<td>Simple and consistent mapping to a simple RESTful web API that is compatible with the Arrowhead translator architecture.</td>
</tr>
<tr>
<td>Service loose coupling</td>
<td>Reduction of dependencies between service provider and service consumer to allow independent implementation and evolution of provider and consumer systems.</td>
<td>Minimal amount of necessary application logic or semantic dependency.</td>
</tr>
<tr>
<td>Service abstraction</td>
<td>Only essential data should be included in the service contract. Implementation details should be hidden from the consumer.</td>
<td>Inclusion of essential aspects only. Smallest possible number of parameters required for normal service consumption.</td>
</tr>
<tr>
<td>Service reusability</td>
<td>Requirement on logic to be broken down into agnostic functionality that can be reused in multiple contexts.</td>
<td>A simple interaction pattern should be applicable across a wider interface.</td>
</tr>
<tr>
<td>Service autonomy</td>
<td>The service should be in control of their own logic and environment to ensure consistency and reliability.</td>
<td>The OPC UA translator spoke should be able to represent an actual OPC UA client in the capabilities its translated mapping offers.</td>
</tr>
<tr>
<td>Service statelessness</td>
<td>Interaction should be stateless to ensure robustness and scalability.</td>
<td>Translation should satisfy a mapping that provide a stateless interaction with the client.</td>
</tr>
<tr>
<td>Service discoverability</td>
<td>Requirement that services should have a meta description to allow evaluation of fitness of use, and to enhance reusability and composability.</td>
<td><em>Not directly applicable to this case, since it is already intended to be captured by Orchestration in the Arrowhead framework.</em></td>
</tr>
<tr>
<td>Service composability</td>
<td>Requirement that services are easily reconfigured and combined to tackle more complex problems.</td>
<td>This property is implicitly captured by the translator architecture and its RESTful interface.</td>
</tr>
</tbody>
</table>
3.1.3 Interoperability Layers

In the design of the Arrowhead translator, Derhamy [10] provide references to a description of three interoperability layers made by the HIMSS (Healthcare Information and Management Systems Society) [1]. Their definition involves; foundational interoperability, structural interoperability and semantic interoperability.

**Foundational Interoperability** enables data to be exchanged between two systems.

**Structural Interoperability** enables a shared understanding of the format, syntax or structure of the data.

**Semantic Interoperability** allows meaning to be exchanged by enabling the interpretation and use of transferred data based on a shared understanding.

<table>
<thead>
<tr>
<th>Example</th>
<th>OSI Layer</th>
<th>Interoperability Solution</th>
</tr>
</thead>
<tbody>
<tr>
<td>XML, JSON, CBOR, SenML</td>
<td>Application</td>
<td>Semantic</td>
</tr>
<tr>
<td>HTTP, CoAP, MQTT</td>
<td>Presentation</td>
<td>Structure</td>
</tr>
<tr>
<td>TCP, UDP</td>
<td>Session</td>
<td></td>
</tr>
<tr>
<td>IPv4/IPv6</td>
<td>Transport</td>
<td>Foundation</td>
</tr>
<tr>
<td>Ethernet, IEEE 802.15</td>
<td>Link</td>
<td>Physical, Network, Gateways</td>
</tr>
</tbody>
</table>

Table 3.2: OSI layers vs interoperability layers vs solutions

When assessing internet standards, it becomes apparent that their placement in the OSI stack also categorize which interoperability layer they belong to. IoT protocols like HTTP, CoAP and MQTT, along with their respective transport-layer protocols (TCP or UDP) form the foundation layer. Their various payload serialization formats (XML, JSON, CBOR, etc.) form the structure layer. The application layer, which is the uppermost OSI layer correspond to the semantic layer.

3.1.4 Translation Transparency

In terms of interoperability, translation transparency means eliminating the need for application logic to be included in the translated interface. This is necessary to preserve loose coupling with the translator service as the (third-party) consumer might not know if translation might occur at design-time.

This is also the main issue about gateway middleware, as established in the related works-section. Because they lack transparency, there is a heavy leakage of endpoint semantics present, which often forces the consumer to understand both the gateway interface and the endpoint interface. As this creates a semantic dependency, interoperability solutions such as gateway middleware only tend to move the problem rather than solve it.

Summarily, for the solution presented in this thesis to avoid the pitfalls of gateway middleware, a lot of thought is put into avoiding semantic dependencies so as to preserve the translation transparency of the pre-existing Arrowhead translator.

1Structural interoperability also ensures that semantic interoperability is preserved at the data field level.
3.2 OPC Unified Architecture

OPC UA combines multiple features in its specification, and can be summarized as a communication protocol with a semantic vocabulary based on an information model.

To interact with an information model hosted by a server, the communication protocol defines 37 services which may be invoked by the client through service calls. Service calls are conveyed over a service session, which is the communication context sustained between client and server. Each session is in turn based on a secure channel, which maintains the integrity and confidentiality of the interaction.

3.2.1 Nodes & References

The base component of the information model is an object-oriented entity called a node, which is identified by its node-id. Nodes are interconnected by references, which help contextualize and link together their associated data in hierarchical trees or horizontal mesh graphs. Every node conforms to a node class which specifies the attribute set of the given node (see fig. 3.2.2).

There are two groups of node classes: 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.

Figure 3.2: In the semantic perspective, references represent a relationship with implicit semantics in between two nodes, but are in practice stored as list entries on their source nodes.

The references that connect node pairs together are also representing a semantic relationship. This is the main reason why nodes themselves are just as important as the data they contain. References help form semantic structures that model the interrelations of complex data. By definition, a reference involves a total of three nodes: a source node, a target node and a ReferenceType-node that contain a type-description of the reference instance itself. Reference instances are stored inside their source nodes, which makes the corresponding node-id an implicit address under which a list of references may be found. Furthermore, to illustrate the semantic direction of the reference instance, reference instances also include an isForward-boolean.

\[ \text{node-id as defined in OPC UA consists of a name and a namespace. The name can be an int, a string, a GUID or a byte-string. The namespace can be an explicit URL or an int to implicitly represent a URL.} \]
3.2.2 Nodeclasses

OPC UA define a set of 8 different nodeclasses; Object, Variable, Method, ObjectType, VariableType, ReferenceType, DataType and View. Furthermore, these are also semantically represented in OPC UA parametrization as an enumeration of that sequence. Distinguishing access classes from type classes, the access classes usually make up the majority of nodes in an address space, and represent the core body of data. The type classes represent node/reference metadata, and help contextualize node/reference instances.

**Object** References variable- and method-nodes as its components.

**Variable** Represents a variable, and container for a variable value.

**Method** Represents a method. When called by a client it returns a result.

**ObjectType** Defines a semantic for Object-nodes. Also, depending on if other component nodes (linked through references) are exposed beneath it or not, an ObjectType-node may expose a structure that is present on each (Object-node) instance of this type. An ObjectType-node with an underlying structure is known as a "Complex" ObjectType - as opposed to a "Simple" ObjectType which is without an underlying structure.

**VariableType** May also be simple or complex. VariableType-nodes may describe engineering units of values, or other characteristics of a variable. Complex VariableTypes (VariableTypes with substructures) define complex variables. This is analogous to defining variable structs, or exposing variables that are inherently associated (e.g. a variable that average the value of other variables). This is not to be confused with structured data types, where structured data of a single type is enclosed in the value-attribute of a single variable.

**ReferenceType** Reference instances themselves are not independent, addressable objects. Instead, they are contained within a node as a collection of data. This way, each node carry its own references to other nodes. Every reference is unidirectional, meaning it always points from a source node to a target node. However, there is usually a conjugal reference that points in the reverse direction stored on the target node. Both instances describe a semantic relationship between the node pair, and this semantic is described by a ReferenceType-node.

**DataType** Provides data type definitions for variable nodes. The set of data-types exposed by an address space are all encompassed by the built-in data types defined by the OPC UA specification. The only exception is the structured data-type, which is not pre-defined by OPC UA. Instead, it is up to the host to define their own encoding, which allow for the definition of custom data type encodings.

**View** To help filter nodes when browsing large address spaces, a node of the class View is used. It can be regarded as a folder that organize nodes associated with a certain context, and may be used as an entry point when browsing.

---

5 The access classes are: Object, Variable, Method and View.

4 The type classes are: ObjectType, VariableType, ReferenceType and DataType.

5 The attribute set of all variable-nodes include the value-attribute. The data stored under that attribute represents the variable value of a variable-node.

6 The reference and its conjugal counterpart are always of the same reference type. In concrete terms, this means that their semantic definition is identical. What separates a reference from its conjugal twin is the direction in which their semantics are pointing. Since the two nodes with conjugal references will always define themselves as the source node, the semantic direction is distinguished by defining an "IsForward"-boolean. This way, the reference direction is also semantically specified. Hence, every node may be completely contextualized by its own references as long as every reference has a conjugal counterpart.

7 Specifically, DataType nodes provide the semantics for how the data type of a variable-node value is to be interpreted.
Figure 3.3: The OPC UA node classes and their respective attributes.
3.2.3 Attributes

All the data contained by nodes (except for reference instances) are stored as attribute values. Depending on what node class a node belongs to, it has a different set of attributes. In total, there are 22 different attributes that may be exposed by a node:

<table>
<thead>
<tr>
<th>Attribute</th>
<th>ID</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NodeID</td>
<td>1</td>
<td>Server-unique identifier for the node. Is composed of a namespace index and a node name.</td>
</tr>
<tr>
<td>NodeClass</td>
<td>2</td>
<td>Integer enumeration specifying what node class the node belongs to.</td>
</tr>
<tr>
<td>BrowseName</td>
<td>3</td>
<td>Used to identify the node when browsing, and is composed of a namespace index and nonlocalized string. Is of the data type QualifiedName.</td>
</tr>
<tr>
<td>DisplayName</td>
<td>4</td>
<td>Name to be displayed in a user interface. Is of the data type LocalizedText.</td>
</tr>
<tr>
<td>Description</td>
<td>5</td>
<td>Textual description to be displayed in a user interface. Also of the data type LocalizedText.</td>
</tr>
<tr>
<td>WriteMask</td>
<td>6</td>
<td>Specifies which attributes of the node can be written to by an OPC UA client. Is of the data type UInt32.</td>
</tr>
<tr>
<td>UserWriteMask</td>
<td>7</td>
<td>Specifies which attributes of the node can be written to by the OPC UA client currently connected to the server. The returned value of this attribute will therefore depend on the currently connected client. Is of the data type UInt32.</td>
</tr>
<tr>
<td>IsAbstract</td>
<td>8</td>
<td>Boolean-value that specifies whether the type-node can be used directly by instances, or if it can only be used as supertype to other type-nodes in a type-hierarchy.</td>
</tr>
<tr>
<td>Symmetric</td>
<td>9</td>
<td>Boolean-value that specifies whether or not the semantic of a reference is the same in the inverse direction.</td>
</tr>
<tr>
<td>InverseName</td>
<td>10</td>
<td>LocalizedText-value that specifies the semantic of the reference in the inverse direction if the reference is not symmetric.</td>
</tr>
<tr>
<td>ContainsNoLoops</td>
<td>11</td>
<td>Boolean-value that indicates if the nodes in a view spans a hierarchy without loops or not, when following hierarchical references.</td>
</tr>
<tr>
<td>EventNotifier</td>
<td>12</td>
<td>Bit-mask that specifies whether the node can be used to subscribe to events, and whether the event history is accessible/changeable.</td>
</tr>
<tr>
<td>Value</td>
<td>13</td>
<td>The actual variable value of a variable-node. The type of value stored under this attribute is specified by the DataType, ValueRank and ArrayDimensions attributes.</td>
</tr>
<tr>
<td>DataType</td>
<td>14</td>
<td>NodeId of the DataType-node that defines the data type of the value attribute of the current Variable-node.</td>
</tr>
<tr>
<td>ValueRank</td>
<td>15</td>
<td>Int32-value specifying the dimensions of a variable-value array.</td>
</tr>
<tr>
<td>ArrayDimensions</td>
<td>16</td>
<td>Optional UInt32-value that specifies the size of each array dimension.</td>
</tr>
<tr>
<td>AccessLevel</td>
<td>17</td>
<td>Bit-mask indicating whether the current value of the value attribute is readable and writable as well as whether the history of the value is readable and changeable.</td>
</tr>
<tr>
<td>UserAccessLevel</td>
<td>18</td>
<td>Same as access level, but with user access rights taken into account.</td>
</tr>
<tr>
<td>Minimum Sampling Interval</td>
<td>19</td>
<td>Optional attribute that specify how fast the OPC UA server can detect changes of the value attribute. (This is relevant when the values are not directly managed by the server, e.g., when the node correspond to a third-party sensor that the server is polling.)</td>
</tr>
<tr>
<td>Historizing</td>
<td>20</td>
<td>Boolean value specifying whether the server stores history for the variable value.</td>
</tr>
<tr>
<td>Executable</td>
<td>21</td>
<td>Boolean value specifying if the method can be invoked.</td>
</tr>
<tr>
<td>UserExecutable</td>
<td>22</td>
<td>Same as executable, but with user access rights taken into account.</td>
</tr>
</tbody>
</table>

Table 3.3: The list of all possible attributes the various node classes may contain, listed with their corresponding (fixed) attribute indexes.

All attribute values belong to a data type defined in OPC UA, and are therefore captured by the built-in DataTypes described in table 3.2.4.
3.2.4 OPC UA Data Types

OPC UA defines 25 built-in data types that form the base of the DataType-node hierarchy. Every data type that is based on the built-in data types reference back to one of these:

<table>
<thead>
<tr>
<th>ID</th>
<th>Name</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Boolean</td>
<td>A two-state logical value (true or false).</td>
</tr>
<tr>
<td>2</td>
<td>SByte</td>
<td>An integer value between $-128$ and $127$.</td>
</tr>
<tr>
<td>3</td>
<td>Byte</td>
<td>An integer value between 0 and 255.</td>
</tr>
<tr>
<td>4</td>
<td>Int16</td>
<td>An integer value between $-32768$ and $32767$.</td>
</tr>
<tr>
<td>5</td>
<td>UInt16</td>
<td>An integer value between 0 and 65535.</td>
</tr>
<tr>
<td>6</td>
<td>Int32</td>
<td>An integer value between $-2147483648$ and $2147483647$.</td>
</tr>
<tr>
<td>7</td>
<td>UInt32</td>
<td>An integer value between 0 and $4294967295$.</td>
</tr>
<tr>
<td>8</td>
<td>Int64</td>
<td>An integer value between $-9223372036854775808$ and $9223372036854775807$.</td>
</tr>
<tr>
<td>9</td>
<td>UInt64</td>
<td>An integer value between 0 and 18446744073709551615.</td>
</tr>
<tr>
<td>10</td>
<td>Float</td>
<td>An IEEE single precision (32 bit) floating point value.</td>
</tr>
<tr>
<td>11</td>
<td>Double</td>
<td>An IEEE double precision (64 bit) floating point value.</td>
</tr>
<tr>
<td>12</td>
<td>String</td>
<td>A sequence of Unicode characters.</td>
</tr>
<tr>
<td>13</td>
<td>DateTime</td>
<td>An instance in time.</td>
</tr>
<tr>
<td>14</td>
<td>GUID</td>
<td>A 16 byte value that can be used as a globally unique identifier.</td>
</tr>
<tr>
<td>15</td>
<td>ByteString</td>
<td>A sequence of octets.</td>
</tr>
<tr>
<td>16</td>
<td>XmlElement</td>
<td>An XML element.</td>
</tr>
<tr>
<td>17</td>
<td>NodeId</td>
<td>An identifier for a node in the address space of an OPC UA Server.</td>
</tr>
<tr>
<td>18</td>
<td>ExpandedNodeId</td>
<td>A NodeId that allows the namespace URI to be specified instead of an index.</td>
</tr>
<tr>
<td>19</td>
<td>StatusCode</td>
<td>A numeric identifier for an error or condition that is associated with a value or an operation.</td>
</tr>
<tr>
<td>20</td>
<td>QualifiedName</td>
<td>A name qualified by a namespace.</td>
</tr>
<tr>
<td>21</td>
<td>LocalizedText</td>
<td>Human readable text with an optional locale identifier.</td>
</tr>
<tr>
<td>22</td>
<td>ExtensionObject</td>
<td>A structure that contains an application specific data type that may not be recognized by the receiver.</td>
</tr>
<tr>
<td>23</td>
<td>DataValue</td>
<td>A data value with an associated status code and timestamps.</td>
</tr>
<tr>
<td>24</td>
<td>Variant</td>
<td>A union of all of the types specified above.</td>
</tr>
<tr>
<td>25</td>
<td>DiagnosticInfo</td>
<td>A structure that contains detailed error and diagnostic information associated with a StatusCode.</td>
</tr>
</tbody>
</table>

Table 3.4: Built-in DataTypes as defined by the OPC UA specification [16].
OPC UA Reference Types

Much like ObjectType, VariableType and DataType, the ReferenceType-node specify the type hierarchy of references. The ReferenceType-nodes describe the semantic categories a reference instance belongs to.

![ReferenceType tree](image)

Figure 3.4: Base of the ReferenceType tree. Every reference in an address space are derived from these.

### 3.2.5 The Address Space

When an OPC UA server instantiates the base nodeset and its own node structures, they form the OPC UA address space of that server instance. The address space is composed only of nodes, and each node is addressed by a server-unique NodeID.

A NodeID is composed of two parts: a namespace-index and a name. The namespace index denotes what namespace (i.e. what original naming authority) a node belongs to. The namespace is stored as a URL in a namespace array, which is supposed to be present in every OPC UA server.

The namespace of the base OPC UA nodeset, which is the foundation of every OPC UA address space, is http://opcfoundation.org/UA/, and is always listed first in the namespace array. Consequently, the namespace index of the OPC UA base nodeset is always 0 by definition.

The OPC UA Base Nodeset

*One thing that is important to note* is that the ID of each built-in data type in table 3.2.4 is not just some arbitrary index assigned to the specification list, but the nodeID-name of actual base DataType-nodes. For example, the node with nodeID-name 1 is the Boolean DataType node in the address space. As such, one realize that much of the specifications mapping is inherent to the base nodeset provided by the OPC Foundation. This base nodeset is expected to be present in every OPC UA server’s address space and is not supposed to ever be changed (except in case of the OPC UA specification being changed). This way, the specification is manifested in the makeup of the address space - contributing to the inherent, self-descriptive nature of the information model.

---

8 The base OPC UA nodeset is available in XML-format at the address: [https://opcfoundation.org/UA/schemas/1.02/Opc.Ua.NodeSet2.xml](https://opcfoundation.org/UA/schemas/1.02/Opc.Ua.NodeSet2.xml)

9 This will become relevant later in the report.
3.2.6 OPC UA Services

While information modeling represent OPC UAs vocabulary, services represent the actual communication of information model content. Therefore, most services are oriented around nodes or their associated attributes and references.

OPC UA define a total of 37 distinct services; 21 of which are used to manage the communication context and its configuration. Only 16 services are used for actual information exchange. The services are divided into functional categories, called service sets, which helps to describe their purpose.

To invoke a service within a server, a client needs to maintain both a secure channel and a session with that server. The secure channel is the foundation of the communication stack, where authentication, authorization, encryption parameters and encoding is determined. The session is the communication context, through which the client may issue service calls. Consequently, the service sets pertinent to discovery, secure channel and session are used for server access and are therefore not directly related to the information model.

The node management service set is used to change the structure of an information model. Since the information model is completely encapsulated by nodes and references, the only services needed are to add or delete them. The view service set is used for navigating the information model structure. Browsing allows clients to explore the references of a node, and to follow reference chains. The query service set is used to search the address space for information.

The attribute service set contain probably the most important services of OPC UA - the Read and Write services. They are used to read the contents of a node, and can be filtered for specific attributes. If the server stores historical data, the HistoryRead- and HistoryUpdate services can be used to retrieve that information as well. The method service set contains a single service; the call (method) service. It is used to invoke method calls on nodes of nodeclass method in the address space. Subscription and monitored items service sets are used in conjunction for OPC UA subscriptions. A subscription is attached to a session, and monitored items specify what information will be subscribed to. They facilitate the subscription functionality in OPC UA (which does not actually follow a true publish-subscribe pattern).\(^{10}\)

\(^{10}\)The OPC UA publish-subscribe services follow a request-response pattern, but delivers a predetermined set of data which is represented by monitored items.

---

Figure 3.5: The services defined by OPC UA.
3.2.7 Service Interaction

Except for services directly related to managing the actual communication context itself, every service request must be made within a session context. Provided the endpoint details are known by the client, the first interaction is to open a secure channel. In this step, the communication stacks of the client- and server-applications establish a connection through which a session can be created. After creation, the session must be activated before it can support service calls for actual data exchange.

After interaction is complete, the client closes both the session and secure channel. If this is neglected, the session and secure channel will timeout, and be closed automatically by the server.

![Interaction diagram](image)

Figure 3.6: Interaction diagram of secure channel setup, session creation and session activation. The principal service calls are marked by the grey area in the diagram. They make up the actual data exchange, rather than communication management.

---

11 Naturally, the services operating outside secure channels or sessions are the services used to set them up or close them. The other exceptions are services related to server discovery, since endpoint details required for secure channel setup may be unknown prior to interaction.
3.2.8 The OPC UA Communication Stack

When implementing any kind of OPC UA client or server, the various parts of OPC UA are layered in its communication stack. There are 3 chief layers; serialization, secure channel and transport.

The semantics and syntax of the OPC UA services are captured by the serialization layer. There, typed data and service call messages are encoded in any of the supported encoding schemes defined by OPC UA; UA Binary, UA XML or UA JSON.

The encoded message is then secured in the secure channel layer (which is configured by the secure channel services), and handed off for transport in the transport layer. Transport protocols supported by the stack are UA TCP, HTTPS, AMQP.

HTTP as OPC UA Stack Transport Isn’t RESTful

As can be noted above, HTTP(s) is already part of OPC UA as a transport protocol. The question then immediately arises why a protocol translation between OPC UA and HTTP is even needed. The broadest answer is that HTTP, as it is used in the OPC UA stack, is not RESTful\(^{12}\) and therefore not compatible with the Arrowhead translator. The main reason for the HTTP-transport not being RESTful is that it does not adhere to a resource-oriented addressing scheme\(^{13}\) or RESTful use of HTTP request methods. In addition, as a transport protocol, HTTP is also deeply embedded into the communication stack itself. It is therefore strictly locked to the syntax and semantics of OPC UA services, which cannot be understood by an unrelated third party. Hence, the HTTP(S) transport in the OPC UA communication stack is therefore of no use to any RESTful translation.

---

\(^{12}\)This was also pointed out by the developers of HyperUA, as a motivation for their solution.

\(^{13}\)RESTful HTTP is centered around abstract resources that can be identified by a URL. Any body of data may therefore be manipulated independently of any intermediary application logic. This is not the case with HTTP in OPC UA.
Implementation Interface

At the top of the stack, applications interact through a stack-specific API. Because the stack is meant to provide an abstract handle on OPC UA entities for the application to interact with, from the application’s point of view, the stack interface semantics are protocol- and encoding-independent. This fact is leveraged to the advantage of the protocol translator presented in this work. By translating OPC UA communication on a level that is not locked to any one encoding or transport, translation will not be limited by any specific endpoint configurations.

Interoperability-Layer Perspective

When put into the context of interoperability layers, the chief layers of the OPC UA-stack can also be compared to both the OSI-stack and the Interoperability-stack.

When reviewing the definition of Semantic Interoperability, which “allows meaning to be exchanged by enabling the interpretation and use of transferred data based on a shared understanding”, it becomes apparent that this is represented by the OPC UA Specification between OPC UA client and server. In essence, the specification itself provides the shared semantics.

<table>
<thead>
<tr>
<th>Interoperability Layer</th>
<th>OPC UA Stack Options</th>
<th>OPC UA Stack Layer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Semantic</td>
<td>OPC UA Services</td>
<td>Application</td>
</tr>
<tr>
<td></td>
<td>Address Space</td>
<td></td>
</tr>
<tr>
<td>Structural</td>
<td>UA Binary, UA XML, UA JSON</td>
<td>Serialization</td>
</tr>
<tr>
<td>Foundational</td>
<td>UA-SecureConversation</td>
<td>Secure Channel</td>
</tr>
<tr>
<td></td>
<td>UA TCP, HTTPS, AMQP</td>
<td>Transport</td>
</tr>
</tbody>
</table>

As stated in the introduction, the main challenge in creating interoperability with OPC UA is to eliminate the interoperability barrier posed by the OPC UA stack. In order to access the address space, which is the structural substrate of the information model, interaction with OPC UA servers is made entirely through OPC UA services. The services, in turn, are communicated through the various standards supported by the stack. Consequently, for translation to occur, all those steps must be bridged in the solution. Furthermore, if there at some point is to be a seamless integration between OPC UA and other systems, semantic translation must also be possible in the future.

The OPC UA services can be regarded as both semantic and structural in nature. They both carry parameters with semantic importance, and are structurally oriented around the OPC UA address space. The application interface to these services is built on the stack API, which manages the communication infrastructure. At the application level, OPC UA semantics/concepts are handled as object-oriented parameters.

3.2.9 Summary

To enable exchange with OPC UA servers without the use of an OPC UA stack (enabling greater interoperability with OPC UA), a holistic translation must translate both structural and foundational aspects of OPC UA. This means that translation must bypass OPC UA transport protocols, secure channel management, serialization and OPC UA services - all in a way that avoid semantic dependencies and preserves translation transparency.

14The only real relevance of protocol/encoding for the application is to establish a communication channel with another OPC UA-based system.
3.3 REST

Representational State Transfer (REST) \[11\] is an architectural style that, like SOA, leverage a set of design constraints meant to increase the interoperability of web-based agents.

3.3.1 Key Features

There are five major principles that form the basis of REST; statelessness, cacheability, layered system, code-on-demand and a uniform interface.

**Statelessness** put constraints on the nature of requests in that it prohibits state-dependencies in requests. A request is required to contain all the information that is necessary for it to be processed, and is not allowed to rely on server-side contexts. For this reason, the client is solely responsible for maintaining its session state in the server.

**Cacheability** leverage the statelessness of requests. Because the server is decoupled from client states, responses may be reused by one or multiple clients to reduce load and improve system efficiency.

**Layered System** help enforce endpoint decoupling by disallowing components to see beyond the immediate layer with which they are interacting. This cross-layer independence is instrumental in minimizing overall system complexity, and can be seen as the decentralizing factor of REST.

**Code-On-Demand** is meant to simplify client interaction by eliminating design-time dependencies by allowing clients to download executable code to extend functionality. The removal of functional, consumer-end requirements allow interaction to become more agnostic.

**Uniform Interface** provide additional reinforcement of interoperability by demanding a uniform interface between interacting components. When independent agents share the same set of interaction rules, the number of interdependencies is reduced.

3.3.2 Uniform Interface Concepts

To help realize uniform interfaces, REST provides the concept of a Resource, which is essentially the target of RESTful interactions. A resource is an abstract representation of any information that can be addressed (be it a document, service or other media). By making requests resource-oriented, REST offer the following concepts for realization:

**Uniform Resource Locator** (URL) is the core handle and address of a resource. It enables a consistent way of accessing a resource.

**Representational Manipulation** allow resources to be consumed in different ways, and promote the access to abstract representations of a resource rather than the actual entity itself (located on the server as a file or concrete object). This enables greater flexibility of presentation and manipulation, as no single format is imposed on the consumer.

**Self-descriptive Messages** are required to accommodate an abstract representation. Since representational access leave more of the resource open to interpretation, the message must contain enough information for how the resource is to be processed.

**HATEOAS** (Hypermedia as the engine of application state) is similar to-, but distinct from self-description in that it provides knowledge on how to interact with the environment in which the resource is located. In other words, to navigate resource trees and gain understanding on how to interact with a server application, HATEOAS allow clients to do so without prior knowledge.
3.3.3 REST in a Service-Oriented Context

There is no real boundary that separate REST and SOA. Many of the principles underpinning either paradigm share the same rationale, and offer the same benefits in managing complex and diverse environments. In the end, both of the REST and SOA architectural patterns provide greater potential for interoperability.

Some analogous concepts are (for instance) uniform interface and standardized service contracts, or layered system and service abstraction/composability where complexity is managed by applying a black-box encapsulation. Statelessness is another example of REST-SOA overlap, and is central to the scalability of diverse and distributed systems - in particular those made up of resource-constrained devices.

Another important feature that is relevant to both REST and SOA is the emphasis on abstract resources as a way of addressing information or services. The abstraction of addressable entities is one of the strengths behind the concept of URLs, which is the basis of the resource-oriented architecture underpinning RESTful protocols such as HTTP and CoAP. As will be explained in later chapters, resource-orientation is a core principle in OPC UA-to-REST translation.

3.3.4 HyperText Transfer Protocol (HTTP)

In many cases, RESTful web APIs are close to synonymous with HTTP (HyperText Transfer Protocol [24]), as it has been the de facto standard protocol for REST APIs and web-based applications for almost two decades.

HTTP is stateless, and not reliant on any session context in its request-response exchange. It also support a uniform interface to RESTful resources. HTTP implements a URL as the resource target address of its request, and supports a set of methods as abstract interactions with the resource that the URL points to. As transport protocol, HTTP use TCP.

Request URL

The Uniform Resource Locator (URL) [33] of a HTTP request points to the resource at which the request is directed, and may represent any abstract entity.

<schema>://<host>:<port>/<path>?<query>#<fragment>

Figure 3.8: URL format for locating resources via HTTP.
Request Methods

To differentiate between what action the request is meant to represent, every HTTP request includes a request-method [34]. The core set of these, based on the CRUD-verbs, are POST, PUT, GET and DELETE.

<table>
<thead>
<tr>
<th>Method</th>
<th>Description</th>
<th>URI Semantic</th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Request that the origin server accept the enclosed entity as a new subordinate of the URI</td>
<td>URI identifies resource to handle enclosed entity. This might not result in a new resource identifiable by a URI.</td>
</tr>
<tr>
<td>PUT</td>
<td>Requests that the enclosed entity be stored under the supplied request-URI.</td>
<td>URI identifies the enclosed entity.</td>
</tr>
<tr>
<td>GET</td>
<td>Retrieve entity (information) identified by the request-URI.</td>
<td>URI identifies the enclosed entity.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Requests that the origin server delete the resource identified by the request-URI.</td>
<td>URI identifies the enclosed entity.</td>
</tr>
</tbody>
</table>

Table 3.6: The semantic meaning of methods.

Idempotence & Safety

An important characteristic of the request method is whether or not they are idempotent [24], (i.e. if the end result is the same for \( N > 0 \) identical requests). The core methods corresponding to this property are the GET, PUT and DELETE methods. A related (but more exclusive) characteristic is method safety. A "safe" method is one that does not cause a state change in the origin server when applied to a resource. The only core method that is considered "safe" is GET.

Collection/Element Resources

In web-development and design of REST-APIs, there is a long-standing feud between supporters and opponents to the concepts of "cool" URIs, the trailing-slash, and distinguishing between collection- or element-resources. The origin of this polarization originate from how resources are inherently organized, and if multiple URIs should be allowed to identify the same resource. A close analogy to this subject can be found in how file-system paths represent files or folders, and whether or not a folder is an addressable resource.

<table>
<thead>
<tr>
<th>Method</th>
<th>Collection, such as <a href="https://api.example.com/resources/">https://api.example.com/resources/</a></th>
<th>Element, such as <a href="https://api.example.com/resources/item17">https://api.example.com/resources/item17</a></th>
</tr>
</thead>
<tbody>
<tr>
<td>POST</td>
<td>Create a new entry in the collection. The new entry's URI is assigned automatically and is usually returned by the operation.</td>
<td>Not generally used. Treat the addressed member as a collection in its own right and create a new entry within it.</td>
</tr>
<tr>
<td>PUT</td>
<td>Replace the entire collection with another collection.</td>
<td>Replace the addressed member of the collection, or if it does not exist, create it.</td>
</tr>
<tr>
<td>GET</td>
<td>List the URIs and perhaps other details of the collection's members.</td>
<td>Retrieve a representation of the addressed member of the collection, expressed in an appropriate Internet media type.</td>
</tr>
<tr>
<td>DELETE</td>
<td>Delete the entire collection.</td>
<td>Delete the addressed member of the collection.</td>
</tr>
</tbody>
</table>

Table 3.7: URI semantics in combination with methods.

When applied to an environment concerning IoT, there are both hazards and benefits to being able to access aggregate information. In the case of interoperability solutions, imposing a distinction for URIs featuring a trailing-slash does create an additional structural parameter to help with translation mappings, but it also impose unnecessary application logic - which is undesirable. A useful rule of thumb in designing a mapping for the translated addressing scheme would be Postel's law [15].

Header & Payload

One interesting aspect of translating OPC UA communication to HTTP is what the HTTP headers should represent. On the one hand, OPC UA communication parameters could be mapped to HTTP header fields to reflect the state of the OPC UA channel/session. This would put third-party clients in closer proximity to the endpoint data, and allow for the response of the endpoint server to be better understood in the case of communication problems or access-related issues. On the other hand, acknowledging that translation transparency (i.e. loose coupling) is required to maximize interoperability immediately raises the point that the client should not need to be aware of OPC UA-related shenanigans when accessing its data. From that point of view, HTTP header fields should only reflect the client’s connection to a virtual server that is representing the translated interface of the endpoint server. Hence, the client should think it is accessing the data directly from the virtual server.

Payload Formats & Hypermedia

Another aspect that becomes relevant when assessing the translation opportunities between OPC UA and HTTP is how to present the exchanged data. After translation has occurred, the semantics directly related to OPC UA services should have been stripped away to only leave the endpoint data. How to re-format this data becomes a difficult question to answer, and without further research into semantic translation, translating the content in any specific way would not necessarily be helpful. In fact, it could easily make interoperability worse, rather than improving it. This topic is therefore left for semantic translation to solve and excluded from the scope of this work.
3.4 Translation Solution Outline

The translation approach taken to translate the entirety of OPC UA has undergone a few iterations, where each has been more stringent than the last. Emphasis has increasingly moved towards favouring RESTful principles as an interoperability goal. Also, translation design has gradually moved closer and closer to an agnostic outlook on the mapping to HTTP, so as to sacrifice some cumbersome OPC UA parameter-semantics rather than forcing them into translation as extraneous application logic. The goal has been to create a generalized, RESTful mapping from OPC UA to REST that is sufficiently abstract to be very easy for HTTP clients to understand.

3.4.1 OPC UA Stack Review

Summarily, OPC UA interoperability is mainly about four things; specification semantics, the address space, services, and their underlying communication context, which encapsulates the various encodings, serialization, encryption and transport.

![Information Model (Meaningful Data)](image)

Figure 3.9: One possible visualization on how OPC UA interoperability barriers can be divided.

**Address Space Essentials**

In summary, the information model constituents exposed by an OPC UA address space can be divided into three structurally distinct data-containing components that are semantically indispensable; nodes, references and attributes. As explained in previous chapters, both references and attributes are exposed through nodes. Hence, to be able to access and manipulate any of those entities, services are naturally oriented around the node as the basic building-block of the address space. Again, it is important to remember that the node is not just a container of data - it also represents a semantic which is characterized by its relation to other nodes. This is why a node instance itself is just as important as the data it contains, and why the nodes in an address space are more than the sum of their parts.

To preserve as much as possible of the core OPC UA functionality after translation has occurred, it must still be possible to:

1. read/write attribute values
2. browse/create/delete references
3. create/delete node instances
An unavoidable consequence of preserving the semantic integrity of node structures is that the translated interface (without semantic translation) is locked to exposing data in a way that will be analogous to the address space of OPC UA. Hence, nodes must still "be a thing" after translation has picked everything else apart. A natural approach to accomplishing this is to expose translated nodes as abstract (RESTful) resources, where their semantic representation is implicit. Also, in order to avoid further semantic dependencies, this must all be possible without requiring the translated interface to involve OPC UA services or context management.

**The OPC UA Stack As A Black Box**

In terms of interoperability- or stack layering, how OPC UA clients/servers are implemented is largely dependent on the stacks they are based on. With OPC UA offering multiple choices for transport and encoding, there are multiple ways of transmitting the same data. Also, since not all stacks support every choice of transport protocol or encoding, it becomes useful to place translation above this layer, and make the translator independent of stack-specific settings. This also helps in encapsulating OPC UA services, so that the interaction of the engaging client can view the translated interface as a black box.

**Structural & Foundational Interoperability is The Goal**

As stated in the introductory chapter, the aim of this thesis is to provide structural and foundational interoperability while leaving semantic interoperability for future solutions to explore. To this end, the solution for structural and foundational interoperability must preserve OPC UA-related semantics as much as possible while avoiding semantic dependencies.

**3.4.2 Translation Interface**

The Arrowhead Translator is intrinsically built with HTTP as its intermediary protocol, and HTTP as a universally (within the scope of the translator) compatible protocol provides a natural starting point for OPC UA translation. Therefore, for all intents and purposes, foundational interoperability is solved if an HTTP client is enough to interact with the interface of the OPC UA translator spoke.

**Structural Interoperability over RESTful HTTP**

For foundational interoperability to serve the structural interoperability layer above it, and for as few constraints as possible to be imposed on the structural and semantic layer, the foundational interoperability need to strictly follow existing design guidelines. To satisfy these, the translation must:

- Follow a RESTful interface
- Be Service-Oriented in architectural makeup by providing:
  - Translation Transparency
  - Service Abstraction
  - Loose Coupling
  - Stateless Interaction
- Provide design freedom of future (RESTful) semantic translation
- Preserve OPC UA core semantics

while bridging the OPC UA stack layers.

---

16 Semantic translation may in the future help make the most of node representations to fully capture their contextual meaning.
3.4.3 Translation Service Mapping Outline

The distinction between OPC UA services maintaining the communication context, and the services that retrieve actual data, becomes more important when surveying what services to translate. As the RESTful translation needs to be stateless, single request must not be dependent on any pre-existing context. Ideally, the management of this context must be decoupled entirely from the client end of the translator. This is essential to not only RESTful interactions, but to service orientation as well. Translation transparency, for instance, is all about removing endpoint dependencies, so the context management of the OPC UA stack is a prime example of what must be encapsulated by the translator. This is achieved by automating secure channel- and session negotiation in the translator spoke on a per-request basis.

When communication-related services are accounted for, there are two major categories of services left to address: subscription and direct data retrieval. These two are oriented around almost the exact same kind of resources. OPC UA subscriptions supply the data changes of monitored items, which are node attributes flagged to notify the subscribing client if their value should change. All in all, the orientation of OPC UA subscriptions revolve around node attributes, but does not follow a stateless interaction pattern. Conclusively, the OPC UA subscription services are not RESTful and therefore not subject to RESTful translation. As data changes of node attributes (monitored items) can be obtained through other OPC UA services, subscription is not included in this work.

The third category, which is about direct data retrieval, is more wide in scope than just node attributes. The service sets for NodeManagement, View, Query and Attribute also involve entire nodes, and the references they contain. However, they are all ultimately oriented around the node, regardless of their purpose. NodeManagement is about adding nodes, or adding a reference between two nodes. The View service set is used to look up reference entries of nodes through the Browse and BrowseNext services. The Query and QueryNext services are used to find nodes and attribute values within the address space. The Attribute service set is perhaps the closest in function to subscription services, as it also give access to node attributes.

Ultimately, all of the direct-access services revolve around the node, and its associated attributes or references. Since they are the only stateless services compliant with RESTful translation, it makes sense to orient translation around the node as its addressable resource. When applied to a RESTful HTTP-translation, the choice of using nodes as addressable resources essentially means representing NodeIDs as URIs in translation. This makes sense for several reasons. First, NodeIDs already act as unique identifiers for every node in OPC UA address spaces (consisting of its namespace index and node name). BrowseNames are also used in some services to identify a node, but NodeIDs, unlike BrowseNames, must be unique within their address space. Being able to unambiguously represent nodes as resources is crucial for a consistent REST-translation. Second, the parametrization of OPC UA services mostly use NodeIDs and attribute indexes as their handle on node-related arguments. Services are therefore already dependent on NodeIDs as an intrinsic parameter. Third, since every node attribute is subordinate to a node, and every reference attached to a source node, they can be represented as subordinate resources to a node-resource. This implies that they can be interpreted by third-parties (devoid of knowledge about OPC UA) within the context of their node. Lastly, since Nodes reference other nodes (reference entries are stored on nodes), nodes may act as hypermedia referencing other hypermedia. This is a core aspect of Representational State Transfer itself in that requests correspond to state transitions when moving from resource to resource, and that resource reference other associate resources.

There is one more service that fit neither the communication context category, nor the subscription category nor the direct data-access categories - the (method) Call service. This service is just as much about invoking an action on a node as it is returning a method result when inputting a method argument. This method will also not be directly incorporated into the translation solution, but discussed as a future possibility.
3.4.4 State-dependence of OPC UA Services

Every OPC UA service except the OpenSecureChannel service is to some degree dependent on pre-existing communication states. By encapsulating the communication context on a per-request basis, state dependence on a secure channel and session can be bypassed.

However, some OPC UA services are still dependent on a preceding exchange of information. Prime examples of this are the QueryNext- and BrowseNext services, which are used on continuation points returned by QueryFirst- and Browse service results. For this reason, some services are not appropriate for a stateless REST-translation.

Reviewing table 3.10 of services, one can see the services considered stateless in [23]. These are essentially the ones permissible in RESTful translation [18]. Only the services that can be invoked without a prerequisite service exchange may be translated into self-contained HTTP requests by the OPC UA translator spoke. The services listed in 3.11 are the ones compatible with the outlined translation.

![Figure 3.10: The services defined by OPC UA. Services marked by * are considered stateless.][1]

![Figure 3.11: The services with inherent support for stateless, RESTful translation.][2]

---

18The exception to this being the previously mentioned services dedicated for managing secure channels and sessions, since they are encapsulated by the translator spoke in every request.
Chapter 4

Design

4.1 Summary of Design Goals

To provide an OPC UA client spoke on the Arrowhead protocol translator, a mapping between HTTP and OPC UA is required, where addressing is oriented around nodes and their respective references/attributes. The translation of the OPC UA communication stack is required to automate and encapsulate the interface to OPC UA services, and to produce an interface to HTTP that feature stateless interaction.

To access the OPC UA address space, the various services related to data exchange must also be covered by the mapping. Unnecessary service parameters must be hidden by the translated interface to provide translation transparency. Neither application logic nor semantic leakage is desired in the translation design.

4.2 Addressing Scheme

By orienting URL of the HTTP requests around the nodes of OPC UA address spaces, their associate attributes or references can be accessed by extensions to the same address. Furthermore, as pointed out previously, a majority of the node/attribute-related parameters can be derived from the node or resource the URL points to. Another benefit is that the URL can help in differentiating between various services, depending on what the URL targets\footnote{There are services that are specific to both nodes, their references, and their attributes. By inferring the type of target, either of these services can be automatically selected.}. This enables the interface to feature the SOA\textsuperscript{1} reusability principle, where the same request can correspond to different services depending on its target.

4.3 Semantic Dependency (and what to do about it)

In an interoperability context, a semantic dependency is for a cross-domain service to be dependent on the semantics or functionality of its endpoint in order to become usable. This essentially forces the consumer to understand the environment in which the service operates instead of providing an abstraction. This is bad for interoperability - not only because it generally negates loose coupling, but because it in a worst-case scenario may create an interface that is too complex to be useful.

High Semantic Dependency

An example of high semantic dependency are the interfaces of protocol gateways, or protocol adapters, where the consumer is required to fully understand the mapping involved for a specific, desired outcome. If the mapping follows some universal standard, there might be a slight alleviation in this dependency. But in cases where the gateway mapping is based on some custom/ad-hoc proprietary protocol, the problem gets even worse. And this is only in terms of request parameters - not consumption of the returned responses.
Low Semantic Dependency

An example of a service with low semantic dependency is RESTful data retrieval, which only requires a URL (and a GET-method) to specify where an abstract entity (resource) can be found. The client has no obligation to understand the resource prior to requesting its retrieval, and may (in the case of HTTP) differentiate the intent of the interaction with a very small set of verbs.

In many cases, semantic dependency is roughly proportional to the prerequisite knowledge needed for a specific service result. This, in an interoperability context, translates to how prerequisite knowledge must cross domain boundaries. As concluded in 2.1.3 gateways (such as HyperUA) still require configuration to be used, and offer very little transparency in their services. Since this configuration is based on OPC UA concepts (AND a custom mapping to those concepts), it introduces semantic dependencies to both the concepts and their mappings.

4.3.1 Semantic Dependency between Interoperability Layers

Another take of semantic dependency (when viewed from an interoperability-layered perspective) is how semantics leak into structural- or foundational interoperability. Foundational interoperability is the exchange of data between two systems. If the exchange require parameters that have semantic significance (i.e. parameters that have to be well understood), there is a semantic dependency in the data exchange.

In OPC UA, there are multiple services that all serve data retrieval, and all of those come with their own parameters. To access a specific kind of information, such as attributes, references or node sets, the consumer must understand how they are different and what service is required for each (as opposed to REST, which features a common data retrieval for any kind of resource, regardless of its data makeup.) Service invocation therefore represent an inherent semantic dependency that must be bypassed.

4.3.2 Complexity of Interaction Patterns

Semantic dependency is also characterized by the complexity of how services are consumed. In the case of data retrieval, the large variation in structural makeup of the OPC UA address space (i.e. the difference between nodes, attributes or references) require a different interaction pattern for each. While nodes are usually navigated by browsing references, and browsing references requires the use of the Browse-service, and the Browse-service require understanding about reference filtering and continuation points, which in turn is completely different to how attributes are accessed - it quickly becomes obvious that the entry point to such a body of information has a far higher semantic threshold to overcome than the simple, uniform interaction pattern of RESTful HTTP.

In essence, requiring any OPC UA semantics to be understood after translation would defeat the purpose of translation itself. If the translated interface would still require a consumer to be aware of OPC UA concepts, it would in a lot of cases be more straightforward to just implement an OPC UA client directly. Hence, if a translated concept cannot be meaningfully rooted in its new environment, it is not really translated, and should be excluded from the translated interface altogether.

4.3.3 In this work

In the scope of this work, semantic dependency is ultimately unavoidable to some degree, due to the complexity of OPC UA. But the intent is to minimize its impact on foundational and structural translation, and reduce the understanding needed to consume the translated interface. In designing a solution, the semantic dependency on structural- and foundational interoperability (data exchange) is here solved by allowing the necessary OPC UA parameters and interaction patterns to be understood in terms of REST/HTTP-related
concepts. In other words, by making the user interface transparent to the client, it can form an understanding on its own terms.

4.4 Limitations Imposed on Design

4.4.1 NodeID Formats

In order to simplify the communication scope, one limitation that is imposed on translation is the omission of extended node IDs. As will be explained in the following section, nodeIDs can be mapped to a URI to form the basis of the node-resource representation. Every node ID contains a namespace-index that represent an entry in the namespace-array of a server’s address space. In extended node IDs, the namespace-index is replaced by an explicit URI (bypassing the namespace-array). This is a generalization made by OPC UA to avoid dependencies on the namespace-array, but it does this at the tradeoff of making node-addressation more intricate, and less abstract. For most cases, the namespace-URI is completely irrelevant, and only used to differentiate nodes that happen to share the same nodeID-name. When translating the nodeID, the namespace-index provide a very convenient abstraction in that it allows a complete nodeID to be represented by

/\ns-index>/<name>/

Figure 4.1: NodeID represented as a URI path

which can be encoded as a path in any URL. With the namespace being a URI in itself, representing it explicitly in another URI would not be feasible. Hence, extended nodeIDs does not translate well, and are therefore not supported in this version of the translation scheme. The consequences of this are discussed in section 6.1.2.

4.5 Service Mapping

Secure Channel and Session-related services are automated, but the remaining (stateless) services require a consistent mapping in translation. Ideally, this is done in a way that reduce the number of interdependencies between client and server. In practice, this means applying some degree of abstraction to eliminate aspects that are irrelevant outside the context of application logic, such as redundant (post-translation) service parameters in OPC UA.

4.5.1 Translation approach to OPC UA service differentiation

By using HTTP as target protocol for OPC UA translation, the service interface used by REST APIs is a natural choice to use as translation interface.

In essence, this means allowing HTTP clients to interact with the translator through requests that are differentiated by URI, request method and payload. Constraining the interface to a minimal set of aspects is useful for keeping it accessible to third parties, and by integrating the service differentiation in a pre-existing interface rather than as an artificial layer of application logic, the REST principle of Layered System is satisfied, along with the principle of Uniform Interface. What this means in practice is that REST clients using HTTP will be able to understand the interface native to RESTful HTTP, without having to adapt to 3rd-party application logic.

4.5.2 Service Translation Alignment

The services that is desired in translation is derived from the list of stateless services.
The previously established requirement for translating these services is making them compatible with a REST API. This is achieved in part by matching these services up with an HTTP request method (POST, PUT, GET or DELETE), and allowing the context in which it is invoked to provide further differentiation. FindServers, GetEndpoints, Browse, TranslateBrowsePathsToNodeIDs, QueryFirst and Read, are all services that involve data retrieval. They are, by this definition, only compatible with GET-requests. RegisterServer, AddNodes, AddReferences, Write and Call are categorized by supplying an instruction or piece of data to be accepted by the server, and are therefore best matched up with either PUT or POST. In line with the above reasoning, DeleteNodes and DeleteReferences are, like the DELETE-method, about resource deletion.

4.6 Service Differentiation within Methods

As each of the HTTP request methods maps to more than one service, it is necessary to provide additional service differentiation that can distinguish a single service from the set of services mapped to each method. Preferably, this must be done in a way that avoids further complexity rather than adding to it.

4.6.1 GET Requests

In the case of GET, the mapped OPC UA services all request data, but of varying character:

**FindServers & GetEndpoints**

FindServers is a service used by clients on discovery servers to obtain its list of registered origin servers. Similarly, the GetEndpoints service is used when an origin server has been found and the client seeks to acquire a set of application URIs that can be used to open a secure channel. Neither FindServers nor GetEndpoints are node-oriented services; they are only about retrieving server metadata. For this reason, these services should be attached directly to the server URI itself (rather than to node-URIs) in order to be resolved in translation.
Browse

In contrast to the FindServers/GetEndpoints services, the Browse service is node oriented. It is not explicitly about reading nodes, but retrieving references pointing from one node to another. This is generally very confusing when getting involved in reference locality, since references themselves are not addressable entities. They don’t have an ID to identify them in the way nodeIDs do for nodes.

References are instantiated in association with the nodes they connect, but OPC UA does not explicitly define how they should be handled internally in the server - only how their instantiation is represented in service parametrization, which include a source node, a target node, and a type node (to represent the reference description). Because reference instances are only accessible through this data, and because two references with conjugal source/target nodes are considered separate (but mirrored) instances, they could be regarded as being stored on their source nodes. To represent this in translation, references should be accessible through their source nodes URI, but not be a separate resource (as this is not reflected in OPC UA). With reference locality being node-oriented, a request to get all references pointing outward from the source node can still be resolved in translation.

TranslateBrowsePathsToNodeIDs

The TranslateBrowsePathsToNodeIDs service is this far the service that is the most ingrained in native OPC UA, and is very difficult to provide a meaningful structural translation for without semantic dependencies. Moreover, representing this service as a GET-request is particularly problematic since the GET-request’s lack of payload would force application logic to be present in the URL. For these reasons, this service will be omitted in translation in this work.

QueryFirst

QueryFirst is a service dedicated to finding data in an address space, but is (like TranslateBrowsePathsToNodeIDs) tied to semantic dependencies and would require imposing application logic on the URL. It is therefore also not included in this translation.

Read

Read is perhaps the most useful and interesting of the data-retrieving services. The Read-service is dedicated to reading attribute values of Nodes, which are all tied to their own semantic meaning. However, unlike TranslateBrowsePathsToNodeIDs, and QueryFirst, the Read-service has no semantic dependencies in structural interoperability - meaning that there is no knowledge of OPC UA required to perform a Read-request.

Differentiation of Services (GET)

Summarily, unifying the OPC UA services associated with a GET-request, involve differentiating FindServers, GetEndpoints, Browse and Read. TranslateBrowsePathsToNodeIDs and QueryFirst are omitted because of their semantic dependency on interoperability, and cannot be translated without semantic translation being solved.

To invoke FindServers and GetEndpoints, a GET request should be made to a generic server resource, with a URI that is supplied for this specific purpose. The result of this request should be the result returned by the OPC UA services, but with structural and foundational information being filtered out so as to apply abstraction on top of OPC UA services while preserving only the meaningful (semantic) data.

To invoke a Browse-request, which returns a set of references, a GET request should be made on a node. With references being represented as collection-like data, they are not

---

2 Naturally, this would also have excluded QueryNext if it had been stateless.
3 The concept of a node is encoded in its resource-representation, and attribute indexes are represented as its sub-resources. Therefore, nodeID and attribute indexes are considered structural in this case.
addressable but their locality and existence still tied to that of their source node. In other words, they are still removed when the node is, and they are not treated as an independent resource. For Read-requests to be differentiated from Browse-requests, the Read-request should be invoked when a GET request is made on any of the nodes attributes. Since attributes have an addressable, consistent index, they can be consistently represented as sub-resources, which would enable per-attribute access. This has huge benefit for resource-constrained devices that only have interest in single attribute values, and therefore would want to keep their communication overhead at a minimum.

4.6.2 PUT & POST Requests

The PUT and POST requests are both about writing information. The distinction between them that becomes important here is that PUT is about creating or updating entire resources while POST is about submitting subordinate data to them. This difference is used further to differentiate between the OPC UA services pertaining to writing information.

RegisterServer

Complementary to FindServers in service discovery, RegisterServer is meant to add a server to the server repository of discovery servers. Since the service translation of FindServers should be limited to a GET on a server URL representing the server repository, the RegisterServer service should only be about incrementing that repository with single entries. Therefore, RegisterServer should be invoked through POST, with the server repository as addressed resource.

AddNodes

This is perhaps (apart from simple attribute read/write) the most straight-forward translation mapping, as it touches on the core idea of representing nodes as resources. Since the node is to be considered an independent resource with its own URL, adding a node in OPC UA is analogous to creating an abstract resource in REST. This is done with a PUT-request, where the request-URL is submitted as the new node URL.

Semantic Implications of Adding Nodes

As was pointed out in earlier chapters, bridging the access to- and manipulation of the address space is the point of the translation here. As nodes form the backbone of the OPC UA address space, they are not possible to exclude in translation. Hence, despite the cost in semantic dependency this brings with it, being able to add or delete nodes is just as important as being able to read or write attributes.

Because OPC UA nodes are inherently associated with a set of attributes, when nodes are added, their attribute sets are automatically added as subordinate resources. With the attribute set being dependent on the nodeclass index, this information is essential for being able to translate an addnode-service invocation at all.

All in all, to add a node, there are three other semantic dependencies on the addnode service-parameters. Nodes in OPC UA address spaces are not allowed to be isolated, meaning they must always be reachable through at least one reference from another node. When adding nodes in OPC UA, this is solved by including a mandatory parent-node along with a mandatory referenceType-node (used for reference instantiation) in the addnode-parametrization. At first glance, this might appear as another semantic dependency, but since the parent-node is addressed by a nodeID, its translation is representable in terms of its RESTful URI. Hence, the semantic dependency is slightly mitigated as the nodeID-parameter may at least be contextualized in terms of a RESTful URI. The same principle applies to the mandatory referenceType-node which can also be represented in terms of its
The third, and last, semantic dependency to be associated with adding nodes, is the circumstantial typeDefinition parameter. If the added node is either an Object-node or a Variable-node, it may imply instantiation of node sub-structures (properties). In practice, this determines if more nodes will be added automatically in junction with this one to reflect the model structure of the template-node. The typeDefinition parameter therefore carry an essential meaning for OPC UA information modeling. Ultimately, when making the argument to include the addnode-service, the question has to be asked if there is any actual difference between writing data to the attributes of a node, or adding a node to a node-structure. The only real difference is how their data is structured. Hence, from a structural standpoint, the data they represent are equally important even if the attribute data fields are less complex.

So... AddNodes?
Yes. AddNodes.

AddReferences
The translation approach of AddReferences can be reached by extrapolating the principles worked out this far surrounding Nodes and their REST representation. As a reference is not an addressable entity in itself, but a subordinate piece of information contained in a node, adding a reference correspond to a POST-request. As previously explained, a reference instance is characterized by three pieces of information; its source node, its target node, and its referenceType-node. It also require two semantically significant pieces of data; the (semantic) direction of the reference and the node class of the target node. Since all of these are mandatory parameters in the AddNode service, they cannot be abstracted. All in all, a reference entry would therefore have 4 pieces of data tied to a node-resource (with the node-resource URL being the source-node address).

Write
Analogous to references being entries in a node-resource, the values of single attributes are stored as separate resources (but with subordinate URIs). So where a node URI follow the format

```
/<namespace-index>/<name>
```

Figure 4.2: NodeID URL structure for an arbitrary node.

the separate attributes have a URI of the format:

```
/<namespace-index>/<name>/<attribute-index>
```

Figure 4.3: NodeID URL structure for an arbitrary attribute of a node.

Since the Write-service is also about writing to pre-existing attributes, it could be invoked as a POST-request on an attribute-resource (with the resource having a URI of the form ![4.3](#). Hence, the service differentiation for writing data to attributes would be the request URL belonging to an attribute-resource - as opposed to the request URL belonging to the node when adding reference entries.

*As the source node is always the same node that the reference is stored on, its source-node address can be implied by the node-resource URL.*
Call (method)

The Call-service is oriented around nodes of the method-nodeclass, and represent function calls of an object-node. The method-node is only a representation of a method, though. OPC UA specify no back-end implementation, only how the method is invoked through communication by the Call-service. When invoking a method, the call-service involve three important parametrizations; object node, method node and input-arguments. The first two are represented by their nodeID (which in translation becomes their URI). The input/output arguments are based on variable-nodes associated with the method node. By definition, a method-node is called with a set of input arguments and returns a set of output arguments, where its execution may or may not represent an action.

Much like the issue of reference locality, OPC UA only specify the service interface (and how to model the method’s parametrization) - not the actual back-end implementation. It is therefore ultimately up to the the server to decide what the method does internally. When comparing the call-service to other services featured in translation, it becomes apparent that Call is the only service to support a possible invocation of an action (as opposed to just reading/writing information) on the modeled objects. For example, if an object node represents a valve, the method node may represent an ability to close the valve.

The catch with the Call-service, though, is that it involve a range of possible input-output arguments. While this is in no way mandatory for a method to require, the service itself include a parameter to support it. If the parameter is left null, no arguments are included in the service call. Since both the methods input- and output-arguments are explicitly based on OPC UA semantics, there exist some semantic dependency, which is not desired in translation. If the call-service is to be supported, translation must not involve semantic dependencies. One way to reduce semantic dependency is to disallow input arguments in translation. This way, a call-request may be performed by only involving two node URIs (object and method).

However, it may be questioned that the call-service itself is of a semantic nature in general. The Call-service is not about storing information, and so any translation to a POST-request would require the understanding that the contents of the request payload would only act as input-arguments, rather than actual information to be stored. This is in contrast to the other translated services, which are only oriented around data access. Therefore, the implementation of a translated Call-service is assuming the underlying semantics of the service is understood by the consumer (since it is not strictly about data access). If a method-node represents turning a valve off, it is only useful as long as the consumer understand that this is the actual end result of invoking it. In more concrete terms, the Call-service requires a deeper understanding of what the ingoing and outgoing arguments mean for the result of the service call, which in turn correspond to a semantic dependency.

Conclusively, as the Call-service features semantic dependencies, it is not suitable for structural/foundational translation. Therefore, it is not supported in the current translation.

Differentiation of Services (PUT)

Summarily, the only OPC UA service supported by a PUT-request is the AddNodes-service, as it is the only service to invoke node-creation (i.e. the only service that translate to direct resource creation).

The request-URL of the PUT-request correspond to the nodeID of the created node. To specify the contents and context of the node, a parent-node (URI) is provided, a referenceType-node (URI) is provided (for describing the reference semantic to the parent-node), and a nodeclass index is provided. The nodeclass index correlates to what set of

---

5This enables method-nodes to describe their functionality by exposing information modeled in the address space.
attribute-resources will be added in junction with the node-resource. Finally, the piece of data to specify underlying node structures and default-values for the attributes, is the nodeID (URI) of the type-node (which acts as a modeling template).

**Differentiation of Services (POST)**

Summarily, the services supported by a POST-request are the RegisterServer-service, AddReferences-service and Write-service. To differentiate between them, the request-URI points to a server-resource, a node-resource, or an attribute-resource (which is a subordinate to the node-resource).

The RegisterServer-service is about adding a server to a discovery-server’s server repository. This translates to adding the same server-data to a server-resource that represent the same functionality.

The AddReferences-service is about adding references between two nodes. Since references are virtually stored on their source-nodes, a POST-request targeting the source-node URI will post a reference-entry as its payload. A reference-entry is a unit of information specifying the target-node (URI) of the reference, and a referenceType-node (URI) to characterize the reference semantic.

The Write-service is about writing data to a node’s attributes. Since attributes are represented as resources in translation, a POST-request directed at an attribute-resource will attempt to store its request-payload under the attribute-resource’s URL.

### 4.6.3 DELETE-requests

The DELETE request-method is used to delete REST-resources, where the request-URL identify the resource to be deleted.

#### DeleteReferences

With the established translation scheme, it immediately becomes apparent that (since references are not addressable resources) the DeleteReferences-service should not be oriented around deleting resources, but rather removing reference entries in existing resources. The only way to resolve this is to change the DeleteReferences-service to instead correspond to a POST-request.

#### DeleteNodes

With DeleteReferences reassigned, DeleteNodes is the only remaining service to correspond to deleting a REST-resource. As its parametrization only require a nodeID, there is no payload required for this request.

### 4.7 Service Mapping Summary

The final mapping of OPC UA services include the services; AddNodes, Read, Browse, FindServers, GetEndpoints, Write, AddReferences, DeleteReferences, RegisterServer and DeleteNodes.

<table>
<thead>
<tr>
<th>Operation</th>
<th>RESTful</th>
<th>OPC UA Service</th>
</tr>
</thead>
<tbody>
<tr>
<td>Create</td>
<td>PUT</td>
<td>AddNodes</td>
</tr>
<tr>
<td>Read</td>
<td>GET</td>
<td>Read, Browse, FindServers, GetEndpoints</td>
</tr>
<tr>
<td>Update</td>
<td>POST</td>
<td>Write, AddReferences, DeleteReferences, RegisterServer</td>
</tr>
<tr>
<td>Delete</td>
<td>DELETE</td>
<td>DeleteNodes</td>
</tr>
</tbody>
</table>

Table 4.1: The Service Mapping of OPC UA Services to the RESTful HTTP Intermediary Format

---

6There is an optional parameter in this service to request that all references pointing to this node also be deleted, but since this would create semantic dependencies while still being possible to accomplish with other services, this parameter is omitted for now.
4.8 Type Support

Unlike GET-requests, PUT and POST allow for a request-payload containing plain text. When parametrization for OPC UA services are translated, having to support data-types that cannot be properly exposed in just plain text would mean having to include type encoding schemes in translation.

Since an encoding scheme would impose application logic and, by extension, semantic dependencies it is not supported in this translation scheme. There is no design feature to disallow this in the future, however. A semantic extension could be attached to provide more complete support. Why this would be desired is on the other hand a more difficult question to answer, since the tradeoff of actually supporting complete OPC UA capabilities in REST-translation would produce just another gateway. It is therefore more beneficial in terms of loose coupling to exclude any parameters that aren’t absolutely necessary.

4.8.1 Plain-Text Payloads

Without semantic translation to solve the issue of semantic dependency, imposing encoding schemes for data-type support is undesirable. Until such a solution exist, the most preferable translation scheme is one where REST-clients only have to care about payloads that can be formulated in plain text.

The core parametrization already proposed can rely on URIs and textual representation of integers (where the translation architecture can leverage a minimal payload format for type-interpretation). Specifically, by constraining the REST-payload of some service translations, the translator can deduce what type is inferred by the context in which the request is made.

This limitation, on only allowing types that can be represented in plain text, will ultimately constrain the flexibility of what the translator can do - but the tradeoff is complete independence of data-type translation schemes, and greatly reduced semantic dependency.

Since the OPC UA translation scheme is intended to interoperate with the Arrowhead-translator, the need to stay clear of semantic dependency becomes even more important. The Arrowhead-translator translate HTTP (which is used as intermediary format) to protocols like MQTT, CoAP and XMPP - all of which are text-based. Forcing third-party consumers to accept (what is for them) completely unrelated endpoint semantics is very bad for interoperability.

Summarily, the constraint on only supporting text-based payloads is slightly limiting on overall flexibility, but necessary for good interoperability.

4.9 Request Formats

To summarize, an OPC UA service is invoked when an HTTP-request is made. The request-method and URL determine the service, and extract the necessary parameters (if any are required) from the request payload. The payload support text-based parameters such as URIs or integer-valued arguments.

---

7DELETE-requests also allow for a payload, but this is not required in this translation scheme.
8This is where a complementary translation for semantic interoperability would be required in order to allow type encoding and other semantics.
9It must be reiterated that this is not a permanent choice. If future work would develop a semantic translation scheme with little or no dependency, this may easily be changed in the translator.
10Translation should prioritize ease of access over translator flexibility. Otherwise, one may just as well implement real OPC UA instead.
4.9.1 Addressing Scheme

The request URL to access a node (and/or) attribute will be of the form 4.4:

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

Figure 4.4: Generalized URL format for accessing a server application resource, node or attribute.

Where the nodeID (namespace index and name) is combined into a path-extension as the ns-idx and name components, and appended to the origin servers authority and application-path. When accessing an attribute of a node, the attribute index is appended to the node URL as attr-idx. If the attribute contains an array, single array elements can be accessed by including an array index, as arr-idx.

NodeIDs with non-textual names

A nodeID consists of a namespace-index and a nodeID-name. The namespace-index is an integer (ranging from 0 and up). The nodeID name can be of four different types; numeric, string, GUID or opaque value.

By default, all nodes in the OPC Foundation standard nodeset have names of numeric type, but string is usually a common type.

In order to expose nodeID-names as URIs, the entire nodeID must be representable as a plain-text string, in which case preserving type-information becomes an issue. Involving type-data in the nodeID would (much like in the case of payload parametrization) impose semantic dependencies.

When encoded in OPC UA communication, nodeIDs does involve an integer-valued enumeration field to specify what type the name is meant to be decoded as. It is certainly technically possible to expose the type-enumeration as another URI component when translating nodeIDs to their URI representation, but this would again impose semantic dependency. Whoever would share the URI would also have to explain what the extra enumeration-component was - perhaps even to consumers familiar with OPC UA, since the component usually isn’t explicitly exposed. It would be straight-forward to explain it is there to define a type, but the underlying enumeration also requires to be understood. The simplest and cleanest approach, at least in terms of interoperability and minimal dependency, is to disallow all but string-based nodeID names. The exception would be the numerical nodeID names of the default OPC UA nodeset, since it always has the same namespace-index (ns=0). As nodes in namespace 0 are always of numeric type, they can automatically be interpreted as such.

4.9.2 Payload Formats

For some of the services, translation parametrization is completed by a request payload. The various parameters include URIs, browsenames (strings), nodeclass indexes (integer), and a direction-indicating boolean. All of these are exposed in plain text, with some formatting (JSON, XML, etc.) for retaining parameter structure to help with payload parsing in the translator logic.

11The OPC UA specification use the term identifier, but to avoid confusion the term name is used here with the same meaning.
12In every address space, namespace-indexes correspond to a namespace-URI in the namespace-array. The default namespace of the OPC Foundation is always listed as index 0 for address spaces implementing the standard nodeset.
The services pertinent to implementing a payload are AddNode, AddReference, DeleteReference, Write and RegisterServer. Each require their own set of parameters, and therefore feature different payload formats.

**AddNode-Service**

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

Figure 4.5: URL/Payload parameter format for adding a node with a PUT-request.

**AddReference-Service**

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

Figure 4.6: URL/Payload parameter format for adding a reference with a POST-request.
DeleteReference-Service

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

Figure 4.7: URL/Payload parameter format for deleting a reference with a POST-request.

Write-Service

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

Figure 4.8: URL/Payload for writing attributes with a POST-request.

4.10 Response Formats

When translating the communication channel, request payloads have been shaped to be as semantically light-weight as possible. Parameters have been excluded for interaction to be as easy as possible. This is done to avoid semantic dependency.

Remember that semantic dependency is about service consumption, and that it dictates how much of the end result the consumer must know in advance to using the service (as opposed to what it must know after using the service).

Translating structural- and foundational interoperability is about enabling information to be conveyed from point A to point B by bridging the mode of communication and restructuring the information to its desired target format (encoding, structure, etc.).

From this perspective, the end result (i.e. the actual data payload in the response) should be largely self-contained if semantic dependency is minimized. If structural and foundational interoperability is properly decoupled from semantic interoperability, the semantic interpretation of the response should be an independent issue. This is where a semantic translation should be applied in the future - to provide understanding of the communicated material.

Hence, structural- and foundational translation should only be required to manipulate response structure - not its semantics. Therefore, the contents of a translated OPC UA-service call response should (if anything) only be restructured. The semantics original to OPC UA should be translated separately, after obtaining this response.

4.10.1 Contents of Response Data

The reason why request data is semantically translated is to overcome its semantic dependency. There is no corresponding demand on response data.

In request translation, service call requests are formulated and invoked, with all related parametrization either derived from the new interface or automatically provided by the translator. The corresponding response would be a body of structured data including everything generated by the request - service result, channel status codes, OPC UA data-types, etc.
The question left open by this work is how to treat the response payload. After it has been restructured for easy digestion, there is nothing more that structural or foundational translation can achieve. Ultimately, semantic translation is required for the end result to be properly understood. To keep dependency at a minimum, and preserve the abstraction introduced in this work, semantic translation could possibly follow the nodes-as-resources outline, and exclude any service-related semantics that can't be properly contextualized by a RESTful representation.
Chapter 5

Implementation

The implementation used as a proof of concept for this thesis consists of a java application constructed to perform the designed translation mapping.

5.1 Arrowhead Translator

The Arrowhead translator was designed to implement protocol spokes as half-step translation pairs when moving from source protocol to target protocol. Each spoke would therefore represent a single protocol mapping with the intermediary format. By traversing any two spokes, a mapping is made between any two of the supported protocols. Since there already was a mapping between HTTP and the intermediary format, the OPC UA translator spoke was only required to interface with the intermediary format to consume HTTP-based requests.

5.1.1 Translation Interface

The translation interface is made up of two chief interactions. The first is setting up a translation channel, which is accomplished by initiating a complementary pair of spokes to form a complete protocol bridge. The second interaction is completely transparent, and provides an automatic translation on every request or response that is exchanged.

Spoke Setup

For clients to interact with the translator, they first have to open up a channel for translation. This means having the translator erect a complementary pair of spokes to form a complete protocol bridge. This is achieved by a single request, which (in HTTP) may have the following format:

```xml
<translatorSetup>
  <providerName>ua</providerName>
  <providerType>ua</providerType>
  <providerAddress>opc.tcp://127.0.0.1:48010?param=arg</providerAddress>
  <consumerName>http</consumerName>
  <consumerType>http</consumerType>
  <consumerAddress>http</consumerAddress>
</translatorSetup>
```

Figure 5.1: Arrowhead Translator request payload format (XML) for setting up a protocol spoke-pair that enables transparent translation.

\footnote{In the Arrowhead Translator, the intermediary format is closely related to HTTP, so in terms of translation they are semantically almost the same thing.}
The parameters extracted from the spoke-setup HTTP-request are used to setup the HTTP and OPC UA protocol spokes. When a spoke-creation request has been handled by the system, and a spoke-pair has been created to handle a complete translation step, it returns a translation endpoint in the response payload. The translation endpoint is used as the URL-authority of the HTTP-requests that are to be handled by the OPC UA translator spoke.

```xml
<translationendpoint>
  <id>55301</id>
  <ip>127.0.0.1</ip>
  <port>51783</port>
</translationendpoint>
```

Figure 5.2: Response format for translation spoke setup. Translation endpoint ID, IP and port. The IP and port are used as the URL-authority in the OPC UA-oriented HTTP requests.

### 5.1.2 Intermediary Format Interface

The IF (Intermediary Format) used as interfaced to the OPC UA client spoke consisted of four major components:
- **Path** The URL path of the incoming request
- **Query** The URL query string of the incoming request
- **Content** The payload of the incoming request
- **Method** The CRUD request-method of the incoming request

### 5.2 OPC UA (client) Translator Spoke

The OPC UA client spoke encapsulates the entire functionality of an OPC UA client, and maintains the client-server secure channel and sessions during its execution of translated requests. It acts as an intermediary to serve the translated requests of the translator’s consumer. To minimize semantic leakage, session-setup and maintenance was baked into spoke-instantiation. This way, no session parameters have to be included in request translation, providing further translation transparency and enhancing translation throughput. In situations where a high amount of traffic is present, being able to pipeline multiple requests through a single session is beneficial for performance. After the spoke is closed, the session and secure channel are discarded.

### 5.2.1 Architecture

The OPC UA client spoke consists of three major components; request parser, mapper and OPC UA client stack.

When a request (represented by the intermediate format of the translator) is received by the OPC UA client translator spoke, it is first interpreted by the request parser. When the syntax and semantics are deconstructed, they are handed off to the translation mapper. It is in the mapper that the actual translation mapping is performed, and it outputs a service-call object that can be invoked (without modification) by the OPC UA client stack in the session that was set up along with the OPC UA protocol spoke.

---

2It is worth pointing out here that the communication context between the OPC UA protocol spoke and the OPC UA server is not stateless. As the context is maintained for the duration of the spoke, this must be realized as an unavoidable consequence of translating OPC UA services. However, as pointed out in the related works-section, recent research has highlighted the benefits of stateless OPC UA-interactions. If this would become a feature of OPC UA, incorporating it into the translator would be trivial.
Request Parser

The request parser breaks down the intermediary format into its most basic parts; path, query, payload and method. It is also in the parser that the payload is decoded from whatever format its parameters are represented in. (In the implementation, only JSON is supported. This can easily be extended to include other encoding formats, such as XML.) As these are required for translation mapping, they are parameterized and handed off to the mapper.

Mapper

Service Differentiation

The first priority of the mapper is to determine the kind of service that is to be invoked by the client stack. It uses the request-method of the intermediate format to map to the predetermined set of OPC UA services (see section 4.7).

The request-method may correspond to more than one distinct service. To distinguish among these services, the targeted resource (interpreted through the request URL path) describe whether the targeted resource correspond to a node or an attribute.

If the service mapping is still inconclusive from the method and URL alone, the content of the payload is used to infer further service differentiation. This narrows the mapping down to a single service, which is used to create a service-call object.

Parameterization

With a service-call object in place, all component parts of the intermediate format provided by the request parser are either mapped directly as arguments, or used to make assumptions about them. (For instance, some OPC UA service parameters related to reference-deletion define whether conjugate references should also be deleted. Since this can be achieved by making another request, the translator assume this is not desired. This decision was made in design to reduce semantic dependency in service parameterization.)

Another parameterization restriction in service-call formulation that is a consequence of its RESTful interface comes from the fact that URLs map to single nodes or single attributes. OPC UA services support multiple nodes to be addressed with a single service call. However, this is not supported by the REST-architecture, and including services as a semantic parameter greatly increase semantic dependency.

Since the translation mapping lack a complete semantic translation, many data-types can not be independently represented in request payloads. The parameters that the translator does support in the request payloads are derived from textual representations. Examples of parameters derived from plain-text representations (with or without an enclosing JSON object) are node paths, browse name-strings, nodeclass index (parsed as an integer), reference direction (parsed as boolean).

When posting values to attributes, the data-type is inferred from the targeted attributes index. Since OPC UA define a consistent mapping for each attribute, this can be defined in the translator without the client needing to know about parameter typing and their respective encodings. Hence, a textual representation is sufficient for the translator to interpret it.

---

3 At the beginning of development, it was not yet decided if there would be any query-string parameters involved, so this data is currently unused in translation.

4 The translation scheme also describe a mapping to server discovery resources, but these are not included in implementation as they require further semantic translation.

5 This will not be the case for request-methods that does not support payloads. There, method and URL is enough for a complete service differentiation.

6 RESTs resource-oriented addressation scheme is intended for a single request to act upon a single resource.

7 An exception to this is the built-in OPC UA type of NodeID, which is represented as the URL-path to a resource.
For parameters such as BrowseName that, like the nodeID, include a namespace-index, it is assumed by the translator that these namespaces have the same namespace-index as the node they belong to. It is worth remembering that this assumption is only made when trying to write to attributes (as a way of reducing parameter complexity) - when reading attributes, no assumptions need to be made and they can be returned as-is.

OPC UA Client Stack

The translators OPC UA client is built on top of the Java-based OPC UA client stack released by the OPC Foundation. It implements the functionality provided by the accompanying libraries to manage the session and invoke service calls.

The OPC UA built-in data types are all supported by the client-stack, and it can therefore be used as building-blocks by the mapper to compose complete service-call objects directly usable by the stack. The translators OPC UA client then interact with OPC UA servers through the client-stack API with the pre-assembled service call objects.

5.3 Test Environment

To test the OPC UA client implementation of the OPC UA translator, multiple third-party tools were used. To avoid having to construct an entire, working OPC UA server from scratch, a number of dummy-servers were used.

5.3.1 Node-OPC UA Server

At the beginning of development, an open-source JavaScript-based OPC UA server was used as dummy server. It featured a limited set of services, but was useful for getting started.

5.3.2 Unified Automation Demo Server

A dummy server obtained from the Unified Automation website\(^8\), based on C and C++, was used during the majority of the development period. It featured more services than the JavaScript-based server, but did not support the NodeManagement service set (AddNodes, DeleteNodes, AddReferences, DeleteReferences). It was ultimately replaced by the more complete Open62541\(^9\) OPC UA server.

5.3.3 Open62541

To be able to test the functionality of the NodeManagement-services in translation, the open-source Open62541 server was used as a dummy.

5.3.4 3rd-party OPC UA Client Software

To verify the interaction of the translators internal OPC UA client, a third-party client-tool called UaExpert\(^10\) released by Unified Automation, was used. It allowed the user to browse the dummy-servers address space and read node data.

---

\(^8\)https://www.unified-automation.com/downloads/opc-ua-servers.html  
\(^9\)https://open62541.org/  
5.3.5 HTTP Tools

To interact with the translator through the consumer protocol, and test the translator, two third-party tools were used to perform generic HTTP-requests.

HTTP Requester (Firefox)

The first tool used for this purpose was the Firefox add-on HttpRequester. It allows the user to formulate their own HTTP requests with almost complete freedom.

Postman (Chrome)

For convenience, HttpRequester was later replaced in testing by the Chrome-supported app Postman, which is almost identical in its application to HttpRequester.

5.3.6 CoAP Wheel Loader Testbed

Since the OPC UA translator works as a protocol-spoke that interface with the translator-hubs intermediate format, it is automatically mapped to all of the other supported protocols (in addition to HTTP). To demonstrate this with CoAP, a CoAP-based testbed-setup was provided by Hasan Derhamy, to act as a third-party client interacting with the OPC UA server through CoAP. It sent simple write-requests to node-attributes hosted on an OPC UA dummy-server, and changed the values of variable-nodes on a periodic interval to simulate sensory readings being updated in real-time.
Chapter 6

Evaluation

6.1 Constraints imposed by translation

There are a number of constraints that (in favour of simplicity) limit the flexibility of third-party clients using a different protocol in their access to the OPC UA-domain. The majority of these constraints arise from abstracting OPC UA-specific concepts in translation.

6.1.1 Services

An example of this abstraction is the implicit OPC UA-service differentiation, which removes the need for third-party clients to know about OPC UA services, but at the same time removes full service control. Why this is at all considered a fair trade-off lies in the goal behind this thesis, namely to eliminate the need for third parties to implement an OPC UA stack, or adopt OPC UA-specific semantics.

If third-parties specifically need full service control, and full control over parametrization, using a translator makes little sense, since the semantic dependencies of a full-feature, translated interface would be equal to or larger than the original OPC UA stack. Moreover, if services were to be included in the translated interface, it would cease to be RESTful.

Payloads & Parameterization

To support the adding of nodes and adding/deleting references (which already have been established in this thesis as an indispensable, core functionality) the parameters that are required to do this have to be supplied by the consumer. The reason why this is at all considered an acceptable semantic dependency is because of the central importance of the node-reference structures of the address space. To mitigate the cost of keeping those parameters, they are translated to be as coherent with the new interface as possible. In other words, the preservation of transparency is improved by expressing as many parameters as possible in a way that is not entirely decoupled from the native environment of the consumer (i.e. nodeIDs are expressed as URLs).

In the case of adding nodes, the only payload parameter that cannot be semantically re-contextualized is the nodeclass index. Even though it can argued that this is transparent if interpreted as only affecting the subordinate resources (attribute-resource) of the node-resource, it does require an understanding. However, it is the only parameter in adding nodes that is not understandable in terms of REST.

When adding or removing references, there is one additional parameter to those required in adding nodes, which is the isForward-boolean. While this could be omitted and set to

1It is here understood that merely expressing a parameter as a URL does not remove the need for understanding its semantic meaning, but its structural significance may be considered to be translated.

2The browsename string parameter could be omitted by the consumer and automatically supplied by the translator, so it is not strictly a semantic dependency.
default as true by the translator, it is the second example of a semantic that cannot be re-contextualized in REST. Fortunately, the nodeclass- and isForward parameters are the only exceptions to hundreds of possible parameters across the entire set of services in OPC UA. Compared to a gateway with full parameter support, the semantic dependency is comparatively insignificant.

6.1.2 Data Types
OPC UA support 25 distinct built-in data types, of which the translator supports. To support all of them (in particular the various types of numerical types, such as unsigned integers of different lengths) a standardized encoding scheme (independent of OPC UA semantics) is required.

Omission of Extended NodeIDs
As with much of the parameter-heavy interaction through OPC UA, extended nodeIDs are intended to give flexibility to regular nodeIDs by expressing the namespace explicitly instead of referencing the namespace by the index it has in the server namespace-array. In most cases, expressing the namespace explicitly is never required, since every OPC UA server is expected to maintain the namespace-array with all its supported namespaces. By removing support for extended nodeIDs, the translator can avoid dependency on this semantic.

Encoding Schemes
It is argued in this thesis that the only supported payload data-types are (and should be) text-based, since involving encoding schemes in translation would increase semantic dependency. Protocols such as HTTP, CoAP or MQTT feature text-based payloads with no one specific encoding or data representation. In that sense, they are data-agnostic, and impose no semantic dependencies by themselves. The protocols therefore allow payloads that are purely based on plain-text, without requiring an encoding scheme.

To preserve loose coupling in translation to OPC UA, this feature must be preserved, which in turn mandates that a bare-minimum-payload must be supported.

This is not to say that encoding schemes have no place in translation, they should just not be obligatory for basic communication. For that reason, they are left for semantic translation to implement.

6.1.3 Security
Security has mostly been ignored in designing the OPC UA translator. Security is an integral part of OPC UA’s elaborate framework of secure channels, sessions and service calls. OPC UA define multiple security objectives in its specification [14], which include authentication, authorization, confidentiality and integrity. OPC UA clients and servers can authenticate themselves with X.509 certificates in a Public Key Infrastructure (PKI), or choose to interact anonymously.

Ongoing research (unrelated to this work) has focused on allowing stateless service calls to make OPC UA support more RESTful interactions [23] solved session and secure channel authentication by constraining stateless service calls to an anonymous "session zero." It would use session-ids and secure channel-ids with the value 0 to indicate stateless communication, with no prior handshaking or channel negotiation. The authors of [23] went on to comment that if allowing anonymous clients to manipulate the data model was deemed risky in terms of process safety, it was left to the server whether or not to implement this.

Since the Arrowhead translator (and by extension its OPC UA client spoke) will represent an intermediary agent with its own certification, anonymous communication channels
will be less exposed to potential risks. However, since the Arrowhead translator does not yet support secured communication, a detailed strategy for incorporating features that can guarantee security within the spoke-server communication has been left for the future.

**Security from Local Automation Clouds**

If the translator is deployed in- and confined to a local automation cloud [8], it is already assumed to be operating in an environment that is isolated from external threats - in which case the security demands are not as harsh as if used in a fully exposed environment.
Chapter 7

Conclusion

OPC UA will require flexible integration with other IoT technologies to efficiently cope with the complexity of Industry 4.0. As previous solutions for interoperability (see sections 2.1.2 and 2.1.3) have featured drawbacks with high engineering cost, a solution has here been presented that enables seamless interoperability between OPC UA and other SOA protocols (such as CoAP, HTTP and MQTT). Compared to others, this solution also requires a comparatively low engineering threshold for implementation.

The translation proposed here satisfy SOA-based paradigms, RESTful interaction and an integration with the Arrowhead framework. Its mapping preserves core OPC UA functionality such as the management, browsing and reading of OPC UA nodes. Because of the complexity of the OPC UA service interface, the mapping does not cover all functions of OPC UA.

The mapping from the OPC UA address space requires that nodes are addressed in the URL path or topic name in a specific manner. The namespace index and node name are required for accessing specific nodes, and adding an attribute index to the same path give granular access to single node attribute values.

Combined with standard CRUD methods, a subset of OPC UA services can be invoked from a generic non-OPC UA service interface.
Chapter 8

Future Work

8.1 Expanding Translation Service Coverage

OPC UAs service utility goes beyond resource-oriented data access. The first example of this is the Call-service, which can be viewed as a kind of remote procedure call (RPC), with arbitrary arguments and return values defined by a corresponding MethodNode. To make this transparent in translation would require semantic translation to be solved first.

The second example is for including a support for publish-subscribe interaction patterns. OPC UA already has a strong support for subscription-based services, and also recently released the PubSub-feature in version 1.04 [17]. To completely cover all aspects of OPC UA, this feature is also left to implement.

One aspect that has been largely ignored in OPC UA translation here are the history-related services. The reason for this has been its close proximity to stateful interaction, heavy parametrization, and lack of idempotence. Hence, translating it to simple HTTP requests would impose a heavy semantic dependency and possibly lack the "safety" required of a GET-request.

8.2 Semantic Translation

Exactly what is meant by the term "semantic translation" is a bit vague, but its main purpose would be to increase interoperability in the same way as structural- or foundational translation does. The goal would be to eliminate the semantic dependencies that arise in current solutions to both reduce the amount of required understanding, and expand the functional range of the translator.

8.2.1 Parametrization

One way for semantic translation to increase interoperability would be to assist in filling in parameters for the consumer. Exactly in what way this would best be solved is not discussed here, but one of the goals would be to allow for more than just plain-text parameter arguments, so that any of the Built-in DataTypes of OPC UA would be supported.

8.2.2 Self-descriptive Interfaces

OPC UA is praised for its powerful ability to describe, and model information in terms of a base vocabulary. It offer the building blocks required to define custom data types and extend the base information model of the default OPC UA nodeset. There is currently no other way to fully introspect OPC UA semantics without out-of-band information (i.e. the OPC UA specification). Semantic translation could reasonably mitigate the need for such understanding in the consumer by making the translated interface more self-descriptive.
8.2.3 Parsing of Response Payloads

To make the most of the data contained in OPC UA servers, and better expose meaning in OPC UA address spaces, OPC UA response payloads would have to be translated. Depending on what kind of information the consumer is interested in, this would also be closely tied to formulating the appropriate requests. Semantic translation would therefore not just help parse the end result - it would help the consumer explore the OPC UA interface as a whole, and provide a holistic understanding.

Because of the large variety in payload content, semantic translation could maybe rely on exposing services with an appropriate granularity to various pieces of information, depending on the context.

HATEOAS

In REST, Hypermedia As The Engine Of Application State (or HATEOAS) is a concept that enables decoupling between client and server by allowing the client to interact dynamically with the server interface without a prior knowledge other than a generic understanding of hypermedia. If applied to semantic translation, HATEOAS could help eliminate the need for out-of-band information as required by the OPC UA specification.

8.3 Reverse Mappings & Discovery

The scope of this translation has been limited to the client-side of OPC UA, where non-OPC UA clients can access the content of OPC UA servers. Future work would also involve a mapping in the reverse direction, where OPC UA clients may access non-OPC UA services.

Also, an OPC UA server discovery would have to be implemented for full lookup and runtime binding routines to be performed.

8.4 gRPC

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.

https://en.wikipedia.org/wiki/HATEOAS
Bibliography


[35] W3C. **URL Definition**. 2017. URL: [https://www.w3.org/Addressing/URL/url-spec.txt](https://www.w3.org/Addressing/URL/url-spec.txt) (visited on 10/06/2017).