FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Web Process Control Protocol

A WebSocket subprotocol for automation

BACHELORARBEIT

zur Erlangung des akademischen Grades

Bachelor of Science

im Rahmen des Studiums

Technische Informatik

eingereicht von

Patrick Roland Gansterer
Matrikelnummer 0928924

an der
Fakultät für Informatik der Technischen Universität Wien

Betreuung: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Mitwirkung: Kolleg. Dipl.-Ing. Andreas Fernbach Bakk.techn.

Wien, 30.12.2015

(Unterschrift Verfasser) (Unterschrift Betreuung)

FAKULTÄT
FÜR !NFORMATIK
Faculty of Informatics

Web Process Control Protocol

A WebSocket subprotocol for automation

BACHELOR'S THESIS

submitted in partial fulfillment of the requirements for the degree of

Bachelor of Science

in

Computer Engineering

by

Patrick Roland Gansterer
Registration Number 0928924

to the Faculty of Informatics
at the Vienna University of Technology

Advisor: Ao.Univ.Prof. Dipl.-Ing. Dr.techn. Wolfgang Kastner
Assistance: Kolleg. Dipl.-Ing. Andreas Fernbach Bakk.techn.

Vienna, 30.12.2015

(Signature of Author) (Signature of Advisor)

Erklärung zur Verfassung der Arbeit

Patrick Roland Gansterer
Hauslabgasse 36/4, 1050 Wien

Hiermit erkläre ich, dass ich diese Arbeit selbständig verfasst habe, dass ich die verwendeten Quellen und Hilfsmittel vollständig angegeben habe und dass ich die Stellen der Arbeit - einschließlich Tabellen, Karten und Abbildungen -, die anderen Werken oder dem Internet im Wortlaut oder dem Sinn nach entnommen sind, auf jeden Fall unter Angabe der Quelle als Entlehnung kenntlich gemacht habe.

(Ort, Datum) (Unterschrift Verfasser)

Abstract

Many embedded devices and programmable logic controller (PLC) provide data via Web services to display them in the browser.

These Web services usually must be polled to get the latest values which introduces a big overhead in communication. OPC UA solves this problem, but requires a high amount of resources as shown by a JavaScript-based OPC UA client for Web browsers. In comparison to that, webMI meets the browsers' needs, but uses plain HTTP communication.

The protocol introduced in this work fills the gap between them by defining a protocol over WebSockets, which can be used directly in the browser with minimum overhead, to be easily integrated into small devices used in the Internet of Things.

Kurzfassung

Viele speicherprogrammierbare Steuerungen (SPS) bieten mittlerweile Web-Schnittstellen, um Daten im Browser darzustellen.

Diese Web-Schnittstellen müssen üblicherweise gepollt werden, um die Werte darstellen zu können, was einen großen Kommunikations-Overhead zur Folge hat. OPC UA löst dieses Problem, benötigt aber eine Reihe von Ressourcen, wie ein JavaScript basierter OPC UA Client für den Webbrowser zeigt. Im Vergleich dazu erfüllt webMI die Anforderungen der Browser, verwendet aber nur reine HTTP-Kommunikation.

Diese Arbeit stellt ein Protokoll basierend auf WebSockets vor, das direkt im Browser verwendet werden kann und keinen großen Overhead verursacht, wodurch es auch in kleineren Geräten, wie sie im Internet der Dinge vorkommen, problemlos verwendet werden kann.

Akronyme

ANSI American National Standards Institute. 38

API Application Programming Interface. 37, 38

CBOR Concise Binary Object Representation. 5, 6, 12, 16, 36, 37

CORS Cross-Origin Resource Sharing. 4

DCOM Distributed Component Object Model. 6

DDoS Distributed Denial of Service. 9

ERP Enterprise-Resource-Planning. 2

HMI Human Machine Interface. 7

HTTP Hypertext Transfer Protocol. iii, v, 1, 3, 4, 13, 14

IoT Internet of Things. 1, 6, 12, 13

JSON JavaScript Object Notation. 5, 12, 14, 16

JSON-CAPS JSON Call And Publish/Subscribe. 12, 15, 17, 18, 23, 24

MES Manufacturing Execution System. 2

MOM Message Oriented Middleware. 13

NAT Network Address Translation. 7, 8, 41

OPC UA OPC Unified Architecture. iii, v, 1, 2, 6, 14, 25, 35, 36, 38–41

PC Personal Computer. 2

PID Proportional-Integral-Derivative. 2

PLC Programmable Logic Controller. 1, 2, 7, 38

RPC Remote Procedure Call. 13, 14

SCADA Supervisory Control and Data Acquisition. 1, 2, 14

SDK Software Development Kit. 38

TCP/IP Transmission Control Protocol / Internet Protocol. 1, 6, 7, 12, 14

TLS Transport Layer Security. 4, 7, 12

WPCP Web Process Control Protocol. 2, 3, 7, 8, 10, 12, 13, 15, 16, 25, 27, 35–38, 41

XHR XMLHttpRequest. 4

XML Extensible Markup Language. 5, 14, 35

Contents

CHAPTER 1

Introduction

In today's industrial automation, there is a trend to switch most of the communication infrastructure across all layers to Ethernet [20]. This step allows many embedded devices and Programmable Logic Controllers (PLCs) to provide data directly via Transmission Control Protocol / Internet Protocol (TCP/IP) and Web services. Additionally to that, there is a trend from proprietary visualization software to the use of Web-based visualizations.

Unfortunately, most of the interfaces provided by the devices do not match the requirements of modern Web technology required for Web-based visualizations. This leaves a gap between the device and the visualization and introduces additional transformation of the data and complexity.

Typical Web services found on devices usually must be polled to get the latest values, which introduces a big overhead in communication. OPC Unified Architecture (OPC UA) [18] could solve this problem, but the required functionality[1] is quite complex to be implemented in constrained devices. Even the implementation in the browser has been shown to require noticeably amount of resources [16]. In comparison to that, *webMI* [11] meets the browser needs, but uses plain HTTP communication, which is stateless and therefore causes a reasonable amount of communication overhead.

This work tries to fill the gap between all these requirements by defining a protocol over WebSockets, which can be used directly in the browser and does not provide a big overhead to be easily integrated into small devices used in the Internet of Things (IoT). As shown in Figure 1.1, it can be used from the Supervisory Control and Data Acquisition (SCADA) level down to the sensor/actuator level in the automation pyramid.

1.1 State Of the Art

Today, still most of the devices found in industrial automation focus on vendor specific protocols. Even if standards like OPC UA find a broader acceptance and adoption, they are usually still sold separately and are no integral part of today's products.

[1] WebService based OPC UA transport

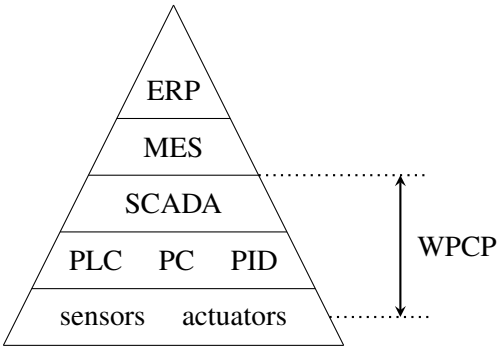

Figure 1.1: WPCP in the automation pyramid

On the other side, there is an increasing interest in the Internet of Things area [21]. More and more devices get connected to the Internet. To ensure interoperability and long period of application, it is important that the communication protocols of them are well understood. This is best done by open and freely available standards.

Visualization for controlling such devices are also mostly built in very specific formats, which do not allow any exchange between different products and often lead to a vendor lock-in situations.

1.2 Goals, Methodology and Structure of the Thesis

The goal of this thesis is to define a communication protocol which fulfills the actual needs of the industrial automation and can be used directly in the browser. It should rely on as many existing technologies as possible to allow easy integration in existing infrastructure.

In the next chapter, Chapter 2, a brief overview of the historically relevant technologies and consently used ones is presented.

The exact requirements for this new protocol and how this relates to existing, well established, protocols are defined in Chapter 3.

Based on these requirements, Chapter 4 defines a generic communication protocol which will be extended by domain specific messages in Chapter 5.

Chapter 6 will then define a mapping between the widely adopted OPC UA standard and WPCP.

Implementations of the presented protocol in different projects are presented in Chapter 7.

CHAPTER 2

State of the Art Concepts and Technologies

Today's computer provide a big variety of different standards and protocols for communication. This chapter gives a short overview and introduction for the ones relevant for this thesis. Some of them are not directly used by WPCP, but are interesting in the historical context.

2.1 Hypertext Transfer Protocol

The Hypertext Transfer Protocol (HTTP) [8] is the basic protocol every Web browser uses to retrieve data from a Web server. In its basic idea it is a stateless protocol where the client sends a request for a resource (e.g. `/path/file.txt`) to the server:

```
GET /path/file.txt HTTP/1.1
Host: www.example.com
```

... and retrieves a response with headers and content back from the server on the same connection

```
HTTP/1.1 200 OK
Date: Fri, 31 Dec 1999 23:59:59 GMT
Content-Type: text/plain
Content-Length: 32

This is the content of the file.
```

Every request and response consist of a set of headers in human readable plain text which are followed by an empty line and actual content. The client has to send a separate request for every resource to the server. Even if the client can reuse an already established connection to send

additional requests it has to wait until the previous response has been transferred completely, which limits the maximum throughput. Such limitations in the protocol lead to HTTP/2 [2] which uses the same semantic, but defines better mechanisms for the shortcomings of earlier versions. All versions of HTTP rely on Transport Layer Security (TLS) [6] to support secure communication.

2.2 XMLHttpRequest

In the early days of the Internet, websites contained only static content and always required retrieving the whole site. To avoid transfer of the same data (e.g. the header of a website) over and over again, browser manufactures introduced the XMLHttpRequest (XHR) object [28] in the scripting environment. This object provides JavaScript functions for sending HTTP requests and retrieving data of the corresponding HTTP response.

For security reasons, the initial version allowed only access to resources from the same server as the calling page. Later the Cross-Origin Resource Sharing (CORS) specification [29] added additional mechanism to securely access resources from other servers, too.

2.3 WebSockets

To overcome the limitation of not supporting bidirectional communication and to reduce the overhead of many small requests done with XMLHttpRequest, the WebSockets protocol [7] has been standardized. To access WebSocket servers, an additional JavaScript object [17] has been specified for the browser scripting environment. Similar to the later versions of the XMLHttpRequest object it features support for CORS [29].

Similar to HTTP it relies on TLS [6] to support secure communication.

Even if it is independent of HTTP, the protocol starts with a handshake, which looks very similar to HTTP, to allow easy integration in existing network structures. The client sends a request to the server, requesting a specific resource, which contains an `Upgrade` header set to `websocket`.

```
GET /chat HTTP/1.1
Host: server.example.com
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Key: dGhlIHNhbXBsZSBub25jZQ==
Origin: http://example.com
Sec-WebSocket-Protocol: chat, superchat
Sec-WebSocket-Version: 13
```

Additional headers starting with `Sec-WebSocket-` prevent the usage of this protocol for malicious tasks and allow the server to adopt itself to the capabilities of the client. A response similar to other HTTP responses indicates support for the WebSocket protocol.

```
HTTP/1.1 101 Switching Protocols
Upgrade: websocket
Connection: Upgrade
Sec-WebSocket-Accept: s3pPLMBiTxaQ9kYGzzhZRbK+xOo=
Sec-WebSocket-Protocol: chat
```

The server informs the client about the negotiated protocol options with the `Sec-WebSocket-*` headers, before the bidirectional communication over the same network connection starts.

2.4 JavaScript Object Notation

The JavaScript Object Notation (JSON) [4] defines a format for representing structured data without a schema as a single text string. It evolved out of the syntax found in JavaScript for defining structured data and therefore supports all of its datatypes. These datatypes are booleans (`true` and `false`), numbers (e.g. `12`, `7.89`), strings (e.g. `"example"`), arrays (e.g. `[1, 2, true]`) and objects which provide a lookup table with strings as keys (e.g. `{"key": 123}`). Since parsing of JSON data is faster compared to other well-known data exchange formats like XML [22], it evolved to a commonly used format for Web services. Implementations for generating and parsing JSON data are available for a wide range of different programming languages.

Figure 2.1 shows a simple JSON object with two attributes. The first attribute with the name `status` has been set to the number `0`, while the second attribute with the name `value` has been set to the boolean value `true`.

```
{"status": 0, "value": true}
```

Figure 2.1: JSON data example

2.5 Concise Binary Object Representation

The Concise Binary Object Representation (CBOR) specification [3] defines a binary encoding for structured but schema-less data like JSON. It defines a superset of the encodable data compared to JSON. For encoding JSON values into CBOR a full set of conversation rules has been defined, which allows easy replacement of JSON data with corresponding CBOR data. The encoding rules follow a simple schema, allowing very simple parser implementations, which reduces the risk of security problems caused by code complexity.

Figure 2.2 shows how the JSON object from Section 2.4 will be represented in CBOR.

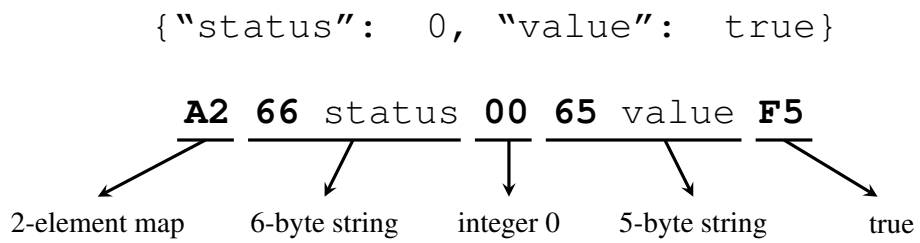

Figure 2.2: CBOR encoding example

2.6 OPC Unified Architecture

The OPC Foundation evolved out of the *OLE for Process Control* standard, which defined vendor independent functions for communication patterns usually found in industrial automation. Since it was based on the Distributed Component Object Model (DCOM) concept of Microsoft, it was only usable on Windows machines, which limited a wider adoption.

Following the DCOM based standard, OPC Unified Architecture (OPC UA) [18] is a new effort by the OPC Foundation to provide a communication protocol between different vendors. Compared to its predecessor, OPC UA does not depend on any technology from Microsoft and does only require a TCP/IP connection. This change allows the implementation of OPC UA conforming products in a wider range of areas. It is now even possible to implement OPC UA servers in PLCs and IoT devices.

OPC UA provides a unified address space with nodes and references between nodes. Every node has a *NodeId*, which acts as a unique identifier. An additonal *BrowseName* for every node allows relative referencing by providing a start node and a list of *BrowseNames*. The *TranslateBrowseNameToNodeId* services provided by an OPC UA server allows efficient resolution of such lists.

Depending on the type of a node, different actions can be executed. For example, nodes of the type *Variable* have services for *Read* and *Write*. For event based implementations *Subscriptions* can be used by clients. They allow the creation of *MonitoredItems*, which define a set of rules, which must apply to get updates on changes. *MonitoredItems* can be used to get updates on value changes of a variable or to receive *Events* associated with a node. *Events* are used by OPC UA to implement *Alarms*.

CHAPTER 3

Design Goals

WPCP was design to fulfill a specific set of use cases. This includes communication patterns between peers and different kinds of network topologies. The following sections give an overview about these points and describe them more detailed.

The most relevant use case for WPCP is the communication between a datasource (e.g. PLC) and a Web browser acting as a Human Machine Interface (HMI). There is a big range of various datasources, which should be all supported equally. At one end of this range there are resource constrained devices like micro-controllers [1] equipped only with a slow network connection. At the other end of this range there are cloud based services, which provide theoretical unlimited processing power and redundant high speed network infrastructure. WPCP uses the Web browser as its main client. Because of that, the effort to implement a conforming client should be minimal in modern Web browsers.

To allow easy implementation of WPCP servers acting as gateways to other systems, existing, well established and broadly known communication patterns should be reused as much as possible.

Where possible, the protocol should be designed in a way allowing the reuse of existing products designed for actual Web technology. This includes the ability to work behind reverse proxies and TLS offloaders.

The protocol should allow the implementation of transparent proxies, which can reduce the load on the original servers and allow connections to servers running behind firewalls using Network Address Translation (NAT) [27].

WPCP must be completely independent of any programming language and specific software technologies (e.g. Java). Only this way it is ensured that the protocol can be implemented in as many devices as possible, without putting effort into porting additional software to the target platform.

[1] They must include a TCP/IP stack and a network interface to support WPCP.

3.1 Network Topologies

WPCP is a client/server protocol. The client, which usually will be a Web browser, establishes a connection to the server and starts the communication. If there are multiple clients, all of them can connect directly to the server, which is illustrated in Figure 3.1. The arrows in the following figures indicate the initiator of a connection. Lines with two arrows stand for equal initiators where each peer may start/setup the connection.

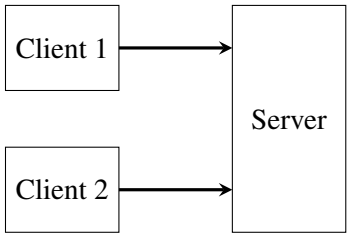

Figure 3.1: Direct Communication

To reduce the load on the server it is possible to insert a proxy between the clients and the server. In that case, all clients are connected to the proxy instead to the original server and the proxy acts as the only client to the server, which is illustrated in Figure 3.2.

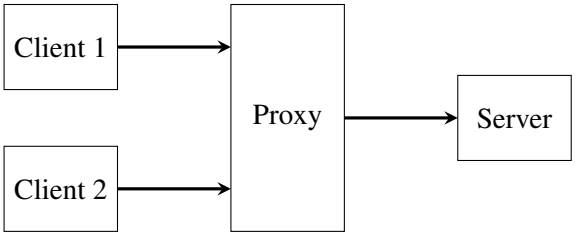

Figure 3.2: Forward Proxy

When used in more complicated network environments it is also possible to stack the proxy servers.

If the different peers run behind NAT enabled firewalls, a proxy can be used to establish a connection. In that case both client and server establish a connection to the proxy, as illustrated in Figure 3.3. The proxy recognizes the connection by a server and can then forward the data from the clients to the server.

Using a proxy in combination with a server behind a firewall has different advantages including, but not limited to the following points:

- On most firewalls, the default configuration can be used, which reduces the overall setup effort.
- If the firewall blocks all incoming connection attempts, there is no exposed port to Internet, which needs constant security monitoring.

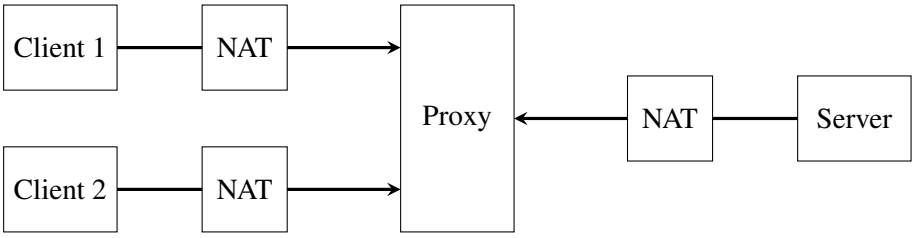

Figure 3.3: Reverse Proxy

- Expensive DDoS prevention mechanisms can be shared across different servers, when they are applied to the proxy instead.

3.2 Redundancy

Support for redundant connection between the peers allow data-loss-free switchover, which is required for high availability applications.

When redundancy is used, peers must be divided into *logical peers* and *physical peers*. *Physical peers* represent different network entities, with different connection endpoints. They usually run on different physical hardware to ensure independent failure states. In contrast to that, *logical peers* represent entities which want to exchange data on an abstract level, independent of the underlying infrastructure. Every *logical peer* needs at least one *physical peer*, which is responsible for the concrete communication.

To take account of the redundancy features one *logical peer* establishes more than one connection to the other *logical peer*. If the connection endpoints do not run on the same physical resource, one connection can get lost (e.g. caused by a crash or network interruption), while the two peers can still exchange data.

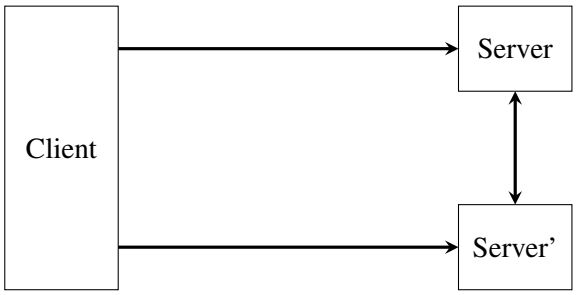

Figure 3.4: Server Redundancy

The two physical peers *Server* and *Server'* represent the same logical server and therefore have to behave in an identical manner. If one of the connections between the client and the server gets lost, the data exchange can be moved to the other connection without loss of any data.

Running a logical peer on more than one physical entity is not limited to servers. Clients and proxies can be built in a redundant fashion, too.

How physical entities representing the same logical unit synchronize their data is out of scope of this specification.

3.3 Communication Patterns

WPCP uses two basic pattern for the communication between the client and the server, which are described in the following sections.

Remote Procedure Calls

This pattern allows the client to execute a routine on the server. When the server finished the execution of the routine, it returns the result of the execution back to the client. Optionally, the server can inform the client about the progress of the execution before returning the final result.

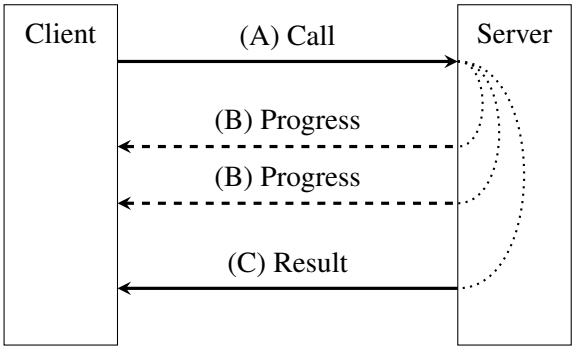

Figure 3.5: Remote Procedure Call Communication Pattern

The abstract message flow illustrated in Figure 3.5 describes the interaction when a client calls a method on the server and includes the following steps:

(A) The client requests the execution of a specific method on the server.

(B) Optionally, the server informs the client about the ongoing progress of the execution.

(C) The client receives the result created by the method executed on the server.

Subscriptions

While the Remote Procedure Calls pattern only allows the client to initiate communication between the peers, the Subscriptions pattern enables transmission of data triggered by the server. To control the data transfer between the peers the server can only provide topics to which the client must register itself first before the server can send data. The client must acknowledge the

receipt of all the data sent by the server. Later, the client can end this server initiated transmission by unregistering from the corresponding topic.

This pattern avoids polling on a topic from the client, since it can register itself for automatic updates on a topic. In that case, no unnecessary data is exchanged between the peers and the client has always the latest state of a topic since it does not need to wait for the next polling request.

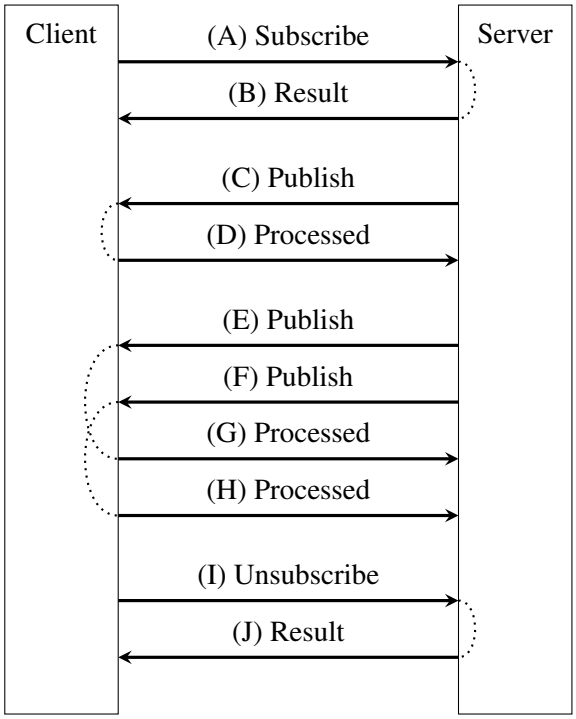

Figure 3.6: Subscription Communication Pattern

The abstract message flow illustrated in Figure 3.6 describes the interaction for publish and subscribe and includes the following steps:

(A) The client subscribes itself to a set of topics.

(B) The client receives an answer for the request in (A).

(C) The server sends updates on the subscribed topics to the client.

(D) The client acknowledges the receipt of the update in (C).

(E) The server sends new data to the client.

(F) The server sends additional data to the client before the previous data has been acknowledged.

(G) The client acknowledges the receipt of the update in (E).

(H) The client acknowledges the receipt of the update in (F).

(I) The client requests the server to stop sending new updates.

(J) The client receives an answer for the request in (I).

3.4 Layered Architecture

To allow easy implementations, the protocol is built in layers. It uses the WebSocket protocol as the foundation, which itself builds on TLS and TCP/IP. On top of the WebSocket protocol this document defines JSON Call And Publish/Subscribe (JSON-CAPS) as a generic communication protocol (see Chapter 4) and Web Process Control Protocol (WPCP) (see Chapter 5), which adds domain specific extensions to JSON-CAPS for usage in industrial automation.

Web Process Control Protocol (WPCP)
JSON Call And Publish/Subscribe (JSON-CAPS)
WebSocket Protocol
optional Transport Layer Security (TLS)
Transmission Control Protocol / Internet Protocol (TCP/IP)

Figure 3.7: WPCP Layers

3.5 Encoding and Transport

The messages exchanged between the peers must be easy to parse, but should be still extensible to support vendor specific extensions. Additionally, the amount of transferred data should be kept at a minimum, even when used without compression. Complex parsing algorithms with high memory requirements or the usage of mandatory compression would make the implementation on small devices harder than necessary.

To fulfill all of these requirements, every message is encoded as CBOR and the message type will be sent as a number instead of using strings for that purpose. CBOR has the advantage of providing semi-structured data like JSON, while still providing a binary encoding of the data. This data format was especially built for IoT devices and features a very easy to implement parsing algorithm, which makes it perfectly suitable for constrained devices.

The protocol has to exchange enough metadata in the first handshake messages, to allow implementation of transparent proxies. Proxies should not need to understand the exact format of every transferred message. By defining the schematic of specific message categories a proxy can use this information to provide caching mechanisms for all servers which conform to the protocol.

3.6 Comparison to other protocols

There is already a big number of communication protocols existing in the wild. The following sections give a brief overview of their design goals and a short description of how they differ from WPCP. Many of them act as Message Oriented Middleware (MOM) [5].

Advanced Message Queuing Protocol (AMQP)

This protocol originated in the financial services industry and has been standardized by ISO/IEC in 2014 [19]. It features a technology independent implementation of a MOM, with solutions for a broad range of middleware problems. It has a broad adoption in the industry, which also includes different open source implementations like *ActiveMQ* and *RabbitMQ*. Free communication libraries are available for all common programming languages. It also defines a WebSocket subprotocol.

AMQP is built to distribute messages across a complex network of producers and consumers. It requires queues for communication between peers, which must be set up on a *broker*. AMQP does not provide native support for RPC and a server does not know which topics are subscribed by clients. So the server must send all messages a potential client might be interested in to the *broker*.

Not knowing which topics clients are interested in would generate many unnecessary transferred messages, resulting in possible inability for communication at all on slow network connections. This disqualifies AMQP for the use in the communication in industrial automation.

Constrained Application Protocol (CoAP)

The Constrained Application Protocol [26] defines a RESTful [9] protocol especially for small, low-power devices. It is similar to HTTP while reducing the size of the exchanged data and reducing the complexity of implementations. CoAP does not require a reliable connection and therefore defines its own mechanism to exchange data in a reliable way. An additional document [15] defines how a client can observe resources to avoid periodic polling.

CoAP does not provide mechanisms for RPC and using it in combination with a WebSocket connection would add unnecessary overhead.

Message Queue Telemetry Transport (MQTT)

MQTT defines itself as a Client Server publish/subscribe messaging transport protocol, which is light weight, open, simple and designed to be easy to implement [1]. Like AMQP it is technology independent, requires a message broker and defines a WebSocket subprotocol, but does not support more complex routing of messages.

The messages used for the communication between peers are simple, which make them ideal for Machine to Machine (M2M) communication and IoT applications, but the missing RPC support and no information about the subscribed topics on the server like AMQP, make MQTT unfitting for the stated design goals.

OPC Unified Architecture (OPC UA)

Standardized 2008 by ISO/IEC [18] this protocol features all communication patterns usually found in industrial automation. The standard has been made freely available in 2015 [24]. Communication happens directly between the client and the server and therefore does not have the requirement for a broker. This form of communication also implies that the server knows exactly which topics a client is subscribed to. It defines a binary and XML encoding for the transferred messages, but with the use of a strict schema, which does not allow easy extensions and requires very specific information about the message format. OPC UA allows the communication via a raw TCP/IP socket or Web services, which both need a complex handshake to establish a secure communication channel. JasUA [16] showed that it is possible to implement a native OPC UA client in the browser, but it does not feature the use of native technologies. Other publications [25] describe additional solutions in the area of OPC UA, too.

Even if OPC UA provides all required communication patters, it is a very complex standard and does not support native Web technologies. The growing count of solution makes implementation of OPC UA compatible solution more easy, but they all have to implement the complex standard, which does not allow light weight solutions.

Web Application Messaging Protocol (WAMP)

This protocol [23] defines a communication protocol using WebSockets for transport. The first and in the meantime outdated version defined a client/server protocol, while the current version 2 requires a broker between the client and server.

Similar to AMQP and MQTT WAMP does not provide information about the subscribed topics to the server, which disqualifies it for the usage in industrial automation, even if it has native support for RPC and uses the native Web technologies WebSockets and JSON for communication.

atvise Web Machine Interface (webMI)

In 2008, *Certec EDV GmbH* created this protocol [11] for their SCADA system *atvise*, which is especially built for the native Web. Built at a time predating the creation of the WebSocket standard, it is built on plain HTTP request using JSON for results returned by the server. This allows the implementation of clients with the browsers `XMLHttpRequest` object. It features RPC and a publish/subscribe mechanism.

Caused by the fact that it uses only HTTP requests, webMI requires a complex session handling and publish behavior to overcome the limitation that HTTP is stateless and does not provide bidirectional communication channels. For more complex data like alarms, webMI limits the possible complexity in comparison to OPC UA, which makes full featured gateways to OPC UA not possible.

CHAPTER 4

Generic Communication Protocol

To make WPCP easier to implement it is split into two layers. The first and lower layer provides a generic protocol for remote procedure calls and subscription as described in Section 3.3. It is called *JavaScript Object Notation Call And Publish/Subscribe (JSON-CAPS)* and makes no assumptions about the area of application and the transferred data. These facts allow a broader adoption and bigger acceptance of the basic layer and can lead to a broader offer of ready-to-use-implementations, which easies the adoption to WPCP.

All additional domain specific functionality for industrial automation is defined in the second and upper layer. Since the lower layer defines already how the peers interact with each other in a generic way, proxies do not need to understand any detail of the upper layer.

The following sections describe the generic layer (JSON-CAPS), while the domain specific layer (WPCP) will be described in the next chapter.

4.1 Message Format

Since the protocol is built on WebSockets the whole communication happens via messages. All transferred messages share the same basic structure, which can be split into the following three basic parts:

Message Type An identifier specifying the kind of the message.

Sequence Number A numeric identifier to group messages belonging to a single action. The peer sending the first message in that group is free to choose this number, but must make sure that responses can be identified correctly, which e.g. is not possible when the same number is used twice before the full response of the previous request has been received.

Payload A list of additional values transporting the actual data. It depends on the *message type* and can be even an empty list.

To exchange this data between the peers, it must be transformed to fit into a WebSocket message. Section 4.2 describes how this is done in detail.

4.2 Encoding and Transport

WPCP uses the CBOR data format to encode messages. CBOR allows the definition of messages without a strict schema and supports all JSON datatypes, which are used natively by the browser, while providing a binary encoding. This binary encoding avoids costly generation and parsing during message exchange.

To transfer a message between two peers the following steps must be executed:

1. The sending peer looks up the index of the message type in the `messages` option of the session.

2. If the message is not a response to a received message, a new sequence number must be generated by the sender. This number can be any number, which has not been used before or for which a response has been received already.

3. The sender constructs an array concatenating the message type index (from step 1), the sequence number (from step 2) and the payload as described by the message type. A message with the message type index `7`, a sequence number `12` and a payload of `[1, true, 2]` would result in `[7, 12, 1, true, 2]`.

4. The array from step 3 gets serialized with CBOR.

5. The CBOR data is sent as a binary WebSocket message to the other peer.

6. The other peer receives the WebSocket message.

7. If the received message is not a binary WebSocket message, the connection must be closed with an error, otherwise the message gets passed to a CBOR decoder.

8. It the decoder is not able to decode exactly one valid array with at least two items, the connection must be closed with an error.

9. The first item in the decoded array must be used to do a lookup in the `message` option of the session. If it is not possible to find a valid message type for that index the connection must be closed with an error.

10. The second item in the decoded array is the sequence number. If the message type defines the message as a response to a sent message, but there is no corresponding outstanding request, the connection must be closed with an error.

11. If the decoded array contains exactly two items, the payload is an empty list. In all other cases the items starting after the second item are treated as payload.

4.3 Message Categories

JSON-CAPS defines messages for calling methods, subscribing topics and handling of the corresponding results. To allow transparent implementations of caching proxies the messages for subscribing are split into three groups depending on the typed of transported data. This leads to a total of five categories, determining the behavior of a message.

ID	Category
G	General
C	Call
E	SubscribeEvent
S	SubscribeSingleValue
M	SubscribeMultipleValues

Table 4.1: Message Categories

General

General messages are used to control the connection between two peers and to response to requests.

All possible *General messages* are described in this work. Applications using this protocol must not define any additional *General messages*, but may define their own set of call and subscribe messages depending on the area of application.

Call

Call messages initiate the execution of a method on the server and get answered with exactly one result. For long running methods the server can send progress messages between the call and result messages to inform the client about already available data.

Subscribe

The client can create a *subscription*, which instructs the server to send updates on the occurrence of events (e.g. the change of a value) to the client. To create such a subscription, the client must send a subscribe message to the server first. Subscribe messages are a special form of the call messages. The server must not send any progress message for a subscribe message and the returned value in the result message must be a non-negative number. Returning zero indicates an error, while any positive number indicates success and is referred to as *subscription id*.

Depending on the kind of updates the server will send after a successful subscribe message the subscribe message belongs to one of the following categories:

SubscribeEvent: The subscription does not have any state and therefore a transparent proxy must not cache any data sent with that subscription id.

SubscribeSingleValue: The subscription is not stateless and the transfered data consists only of one value. This value must be sent to the client directly after a successful subscription has been created and when the value changed. Proxies are allowed to cache the last received value and do not need to create a subscription for every client, if they subscribe to the same topic.

SubscribeMultipleValues: This category is intended for stateful subscriptions, which have to transfer a list of values to the client. They impose some form to the transferred data, but in return allow updates of single items in the list. Proxies are allowed to cache the list, but compared to storing only the latest value for *SubscribeSingleValue* subscriptions a more complex handling is required to keep the list in sync with the server.

4.4 Message Types

The following sections describe all messages, which are defined by JSON-CAPS.

General Messages

The messages belonging to this category are the foundation of JSON-CAPS. They are used to transfer the actual data and are responsible for the correct flow of other messages.

Publish

This message is used to send updates on subscriptions to the client. The client must have created a subscription first, before this message can be sent. One publish message can contain data for different subscriptions at once.

Message Category General

Message Type `publish`

Sequence Number Can be freely chosen by the sending peer, which must ensure that a number is not used twice to ensure that the corresponding `processed` message can be identified correctly.

Payload List of alternating *subscription id* and *publish value*. *Subscription id* is the number returned by a subscribe message as described in Section 4.3. *Publish value* is the actual value sent via the subscription.

Processed

This message is used to acknowledge a publish message. A client must send this message after all the data in a publish message have been processed. It signals the server that a publish was delivered successfully and makes the sequence number reusable for a new publish message.

Message Category General

Message Type `processed`

Sequence Number *identical to the one in the corresponding publish message*

Payload *This message type has no payload.*

Progress

This message can be used to update the client with information about the progress of a call.

Message Category General

Message Type `progress`

Sequence Number *identical to the one in the corresponding call message*

Payload List of alternating *callposition* and *value*. *Callposition* is the position of the corresponding subcall in the call message and *value* is the corresponding value of the item.

Result

This message is used to send the final result of a call to the client.

Message Category General

Message Type `result`

Sequence Number *identical to the one in the corresponding call message*

Payload List of alternating *info* and *value*. *Info* can be an object containing additional diagnostic information (e.g. error message) to a specific subcall or must be *NULL* otherwise, which is also the default value. *Value* is the corresponding value of the subcall.

Cancel Call

A client can use this message to interrupt a pending call. When a server receives such a message it must stop processing of the requested call and returns a result message as soon as possible. The result message of a cancelled call should contain information about the cancellation in the info object. If the sequence number is not known to the server, the message must be ignored.

Message Category General

Message Type `cancelcall`

Sequence Number *identical to the one in the corresponding call message*

Payload *This message type has no payload.*

Call Messages

The messages in this category are used to invoke a method on the server. Every message contains one or more *subcalls*. Every *subcall* is represented via an item in the payload and must be handled as an independent method invocation. Sending one message with three payload items must behave exactly the same as sending three messages with one payload item. A client should send as many *subcalls* in one message, since it reduces the resource requirements and allows the server to do some optimizations (e.g. for locking).

Every message must be answered with exactly one `result` message, which contains the same sequence number as in the call message. After the sending peer received the corresponding `result` message, the sequence number can be used again.

Optionally, the server can send a `progress` message before sending the `result` message. If not stated otherwise, the following methods do not send any `progress` messages.

Ping

To implement a heartbeat functionality which checks the liveness of a working connection the client can call the `ping` method. This method returns the same data, which has been sent in the request. If the transport mechanism provides a more efficient way to check the connection state, the client can use the alternative option.

Message Category Call

Message Type `ping`

Payload Item freely choosable by the sending peer

Result Item Value identical to the payload item

Unsubscribe

This method is used to remove existing subscriptions. After receiving this message a server must not send any additional data for the unsubscribed subscriptions if the reference count reached zero.

Message Category Call

Message Type `unsubscribe`

Payload Item subscription id as described in Section 4.3

Result Item Value reference count of the subscribed item before unsubscribing. *0* means that the sent subscription id is not known to the server. *1* indicates that the last subscription for a topic has been unsubscribed and any allocated resources for that topic can be freed at the server. Any larger value indicates that the subscription id has been used more than once and the server has allocated resources for that subscribed topic. If the invalid subscription id *0* has been sent by the client the server must respond with *0* in the `result` message.

Transfer Session

This method is used to transfer the session to the connection over which this message has been sent.

Message Category Call

Message Type `transfersession`

Payload Item object containing the message type and sequence number of the last valid received message by the client from the server in the session: the message type uses the property name `type` and the sequence number `id`

Result Item Value object containing the message type and sequence number of the last valid received message by the server from the client in the session: the message type uses the property name `type` and the sequence number `id`. In the case of an error *NULL* should be returned.

Subscribe Messages

All messages in this category are use by the client to create subscriptions for topics it wants to get updates for. The difference between the three subscribe message is only the way the data is sent in the publish messages as described in Section 4.3.

Subscribing to a topic is done by calling a subscribe method on the server. Until an `unsubscribe` message has been sent, the server must send updates on that topic to the client via `publish` messages. The exact content of the `publish` messages is described in Section 4.6.

4.5 Sessions

A session is a semi-permanent interactive information interchange between a client and a server. It is required to exchange any data between the peers. Since establishing a session is mandatory, the first message which is sent on a created connection is used to negotiate the details of the session. This first message is called *hello message* and does not have its own message type. Therefore, the message type set in the *hello message* must be ignored by the server and the client can use any value for it. The sequence number follows the rules for calls, where the client can freely choose the value. It is recommended to use 0 as the sequence number and the message type in the *hello message*.

For now, the payload of the *hello message* consist of exactly one object, referred to as *session request options*. The valid properties for this object are defined in the next section. Sending more than one payload item is reserved for future extensions to the protocol and a conforming server must ignore any additional payload items.

After receiving the *hello message*, the server must respond to it with a `result` message using the sequence number of *hello message* and a payload with exactly one object as item, referred to as *session response options*. The valid entries for this object are defined in the next section. Sending more than one payload item is reserved for future extensions to this protocol and a conforming client must ignore any additional payload items.

Session Options

The session options are uses to control the behavior of the session. The client can state its capabilities and preferences in the *session request options* in the *hello message*, while the server makes the final decision about the used values and sends them to the client via the *session response options*.

`messages` (REQUIRED): string-array with the names of all supported message types prefixed with the message category id. The client can send all messages it is able to receive. The server then sends the complete set of messages which are valid in the session for both peers.

`sessionid` (OPTIONAL): opaque unique identifier for the session created by the server. This property is required if the session can have multiple connections. A client sending this value indicates that no new session should be created by the server and the connection should be added to the existing one with the provided identifier. The client must always compare the returned value from the server with the sent value, to check if adding the connection succeeded or if a new session has been created.

`idletimeout` (OPTIONAL): number of seconds a session will survive without an active connection. A value of *0* indicates that the session will become immediate invalid when the server has no open connection left to the client. A value of *-1* indicates that the session will never get teared down. Independent of the returned value the server can still create a new session before the timeout has been reached, if it is unable to recover the last session, which will be indicated with a different sessionid.

Transfer Session

This feature can be used for recovering a session where all connections have been lost to and to support redundancy without loss of data. Both points base on the fact that a session can have more than one connection. In the first case, the connections to a session happen after each other. In the second case there are multiple connections to the same session at the same time.

When more than one connection is used, there is always exactly one *active* connection. All other connections are *passive*. The connection which gets a new session assigned is automatically the active connection. Later connections reusing the same session are in the passive state after the initial hello and corresponding result message. This means that no communication is allowed beside calling the `transfersession` method. The active connection always allows the complete set of messages, including `transfersession`.

The `transfersession` method is used to switch the active one between multiple connections or to activate the newly established connection after a connection lost. To switch the connection, the client has to send a `transfersession` message with the type and sequence number of the last received message on the active connection by the client. The server responses with message type and sequence number of the last message which the server has received on the active connection.

If the `transfersession` call succeeded, the connection over which the call has been executed is the new active connection and all other connections are switched to the passive state. Immediately after receiving a positive result, the consumer must resend any messages which might have been sent but have not been received by the server on the active connection. The server does the same for all message, which have not been received by the client. All passive connections must not be used to transfer any data expect a `transfersession` call, until they become active.

4.6 Subscriptions

The server assigns every subscribed topic a unique subscription id, which is used to match the content of the `publish` messages to their corresponding subscription. If the requested topic is already subscribed or if the topic is an alias of an already subscribed topic, the server can return the subscription id of the existing subscription and increment the reference count of the subscription.

JSON-CAPS supports three kinds of subscriptions, which differentiate only by the data from the server to the client. The following section describes how the data has to be sent by the server in more detail.

Event

This subscription type is used for data without a state. Proxies must not store any published data of this subscription type and are only allowed to forward the received data from the server. What data is sent with the event is completely up to the server.

A subscription of this type could for example be used to send events of pressing a button. For sending the state of the button (pressed or released) the next subscription type should be used.

Single Value

This subscription type is used for stateful data, which has exactly one value all the time. A proxy can cache the latest received publish item and send it to a client, which is requesting the same topic without communication to the original server. What data is sent with the event is completely up to the server. After establishing a new subscription the server must send the actual value to client, even if it has not changed. This is necessary to give the client an initial value.

A subscription of this type could for example be used to exchange the temperature measured by a sensor. The sensor has exactly one temperature at a time, which is only sent to the client when the value changed.

Multiple Values

This subscription type is used for exchanging a list of data items between two peers in an efficient way. It is the most complex type and imposes additional constraints on the format of the

transferred data. Every data item in the list must be an object containing a `key` and `retain` property, which are used to control the handling of the items in the list. Applications using JSON-CAPS have to define additional properties in this object to transfer the actual data.

After a successful creation of a subscription the client has only an empty list and the following algorithm is applied to every received publish item to populate it:

- If the received value is not an object an error is thrown.
- After creating a new subscription, the client needs a mechanism to know when all items which have been active at the moment of creating the subsciption have been transferred. Receiving an object without `key` property indicates that all of these initial items have been transferred to the client. Any additional items which will be received afterwards are updates to the existing list.
- If the received object contains a `retain` property set to *true*, the client must search for an item in its list, which `key` property has the same value as the `key` property of the received object. If such an object has been found in the list, it gets replaced with the received object. Otherwise, the received object gets inserted as a new item into the list at the client.
- In all other cases the client must search for an item in its list, which `key` property has the same value as the `key` property of the received object and remove it from the list if found.

A proxy can cache the current list of data by updating its list with the same algorithm as a client would do. This cached list can then be sent to a client requesting the same topic, without communication with the original server.

A subscription of this type could for example be used to exchange a list of sensors measuring a critical value. If every critical value corresponds to one item in the list a new critical value does not require sending the whole list to the client over and over again. Instead only the new value needs to be transferred, which leads to a more efficient transport.

CHAPTER 5

Domain Specific Messages

Based on the generic communication protocol defined in the previous chapter, this chapter defines domain specific messages for the communication requirements in industrial automation. The format and content of the messages allows an easy mapping to existing datasources. A detailed mapping for OPC UA will be defined in Chapter 6.

5.1 Address Space

WPCP uses a hierarchical graph built with nodes and references connecting two different nodes with each other, creating a parent/child relationship. The graph can have circular references. Every node has a unique identifier for direct addressing. The identifier of the root node is always the empty string. Additionally, child nodes have a `name` attribute, for addressing them in relation to their parent node. Optionally, all nodes can have a `title` defining a human readable name for a node, `description` for a detailed description of the node and a `type` for defining the behavior of the node.

Every node can take the role of a *dataitem* and/or *eventnotifier*. Nodes without a role are only used for structuring the address space. Nodes which are *dataitems* have a value associated with them, which usually can be *read*, *written* and *subscribed*. The *eventnotifier* role marks nodes able to emit events, which can be *subscribed*.

The example in Figure 5.1 shows a hypothetic building with 5 rooms on two floors, each measuring the temperature and the light intensity.

Addressing Schema

WPCP defines two schemas for addressing the various nodes. The addresses are usually used only with the `id` property in the request payload items.

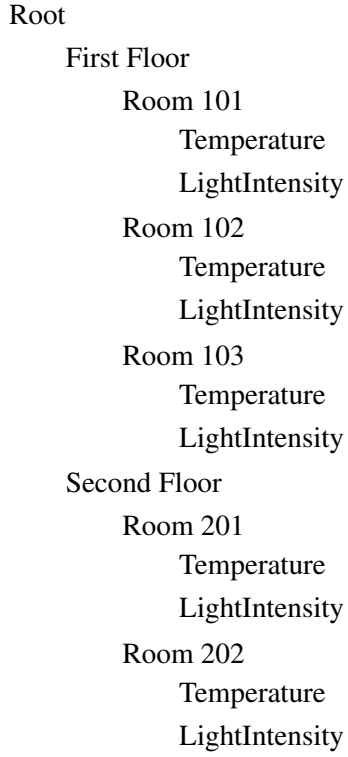

Figure 5.1: Example Address Space

Absolute Addresses If the exact unique identifier of the node is known in advance, this id can be passed as a string to the server. The exact format is specific to the used server. Only the empty string must refer to the root node.

E.g.: `{"id": "GlobalTemparature7"}`

Relative Addresses As alternative the combination of a start node (which must be addressed absolute) and a list of node names can be used to address nodes. To use this schema an array with the starting node as first item and the node names as the following items is passed to the server.

E.g.: `{"id": ["Motor5", "Temparatures", "Case"]}`

5.2 Object Attributes

The two peers handle most of the communication via objects in the payload. This technique allows easy extension, while ensuring full compatibility. Many of the defined methods use the same attribute name for the same or similar functionality. The attribute names used by WPCP are listed in Table 5.1 and will be used again in the following sections.

Key	Description
`acknowledged`	Boolean value indicating if the alarm already has been acknowledged.
`active`	Boolean value indicating if the state of the alarm refers to a situation in which the *alarm* actually exists.
`aggregation`	An identifier used to determine how the values should be aggregated. The valid values are listed in Section 5.5.
`description`	The description of a node.
`endtime`	End of period to read.
`filter`	An object defining a filter which must be applied to the result of the request.
`id`	The unique identifier of a node as described in Section 5.1.
`interval`	Interval between aggregated values.
`key`	As described in JSON-CAPS.
`maxresults`	The maximum number of results which should be returned.
`message`	A textual description of an *alarm* to be displayed in the UI of the HMI.
`name`	The name of a node.
`priority`	The priority of an *alarm*.
`retain`	As described in JSON-CAPS.
`starttime`	Beginning of period to read.
`status`	Information about the quality of the value. A missing `status` key or a falsy value (e.g. `false` or `0`) indicates a *Good* state.
`timestamp`	Number of milliseconds since 1970-01-01 when the value or state changed.
`title`	The title of a node.
`token`	A unique and opaque identifier for use with `handlealarm` calls.
`type`	The type of a node.
`value`	Value of the datapoint. If possible objects should be avoided as type of the value.

Table 5.1: Object Keys

5.3 Messages

The following sections define additional message types, which can be used to meet the domain specific requirements in the industrial automation.

Read Data

This service is used to retrieve the current value of a *dataitem* from the server.

Message Category Call

Message Type `readdata`

Payload Item object with following keys:

- **`id` (REQUIRED)** The unique id of the *dataitem* which should be read.

Result Item Value `null` in case of an error or an object with following keys on success:

- **`status` (OPTIONAL)** Information about the quality of the value. A missing `status` key or a falsy (e.g. `false` or `0`) value indicates a *Good* state.
- **`timestamp` (OPTIONAL)** Number of milliseconds since 1970-01-01 when the value or its state changed the last time.
- **`value` (REQUIRED)** The actual value of the data item.

Write Data

This service is used to set the value of a *dataitem* at the server.

Message Category Call

Message Type `writedata`

Payload Item object with following keys:

- **`id` (REQUIRED)** The unique id of the *dataitem* which should be written.
- **`value` (REQUIRED)** The new value for the *dataitem*.

Result Item Value The boolean value `true` if successful, `false` otherwise.

Read History Data

This service is used to retrieve the historical values of a *dataitem* from the server.

Message Category Call

Message Type `readhistorydata`

Payload Item object with following keys:

`id` (REQUIRED) The unique id of the *dataitem*.

`starttime` (OPTIONAL) Beginning of period to read.

`endtime` (OPTIONAL) End of period to read.

`maxresults` (OPTIONAL) The maximum number of results which should be returned.

`interval` (OPTIONAL) Interval between aggregated values.

`aggregation` (OPTIONAL) An identifier used to determine how the values should be aggregated. The valid values are listed in Section 5.5.

Result Item Value `null` in case of an error or a list of objects with following keys:

`value` (REQUIRED) The value of the *dataitem*.

`timestamp` (REQUIRED) Number of milliseconds since 1970-01-01 when the value or its state changed. If `interval` is requested, the timestamp of the interval must be used instead.

`status` (OPTIONAL) Information about the quality of the value. A missing `status` key or a falsy (e.g. `false` or `0`) value indicates a *Good* state.

Read History Alarm

This service is used to retrieve the historical states of *alarms* from the server.

Message Category Call

Message Type `readhistoryalarm`

Payload Item object with following keys:

`id` (REQUIRED) The unique id of the *eventnotifier*.

`starttime` (OPTIONAL) Beginning of period to read.

`endtime` (OPTIONAL) End of period to read.

`maxresults` (OPTIONAL) The maximum number of results which should be returned.

`filter` (OPTIONAL) A filter used to determine which event should be returned.

Result Item Value `null` in case of an error or a list of objects with following keys:

`key` (REQUIRED) As described in JSON-CAPS.

`retain` (OPTIONAL) As described in JSON-CAPS.

`token` (OPTIONAL) A unique and opaque identifier for usage with `handlealarm` calls.

`id` (OPTIONAL) The unique id of the *dataitem*, which is responsible for the *alarm*.

`timestamp` (REQUIRED) Number of milliseconds since 1970-01-01 when the value or its state changed.

`priority` (OPTIONAL) The priority of the alarm.

`message` (OPTIONAL) A textual description of an *alarm* to be displayed in the UI of the HMI.

`acknowledged` (OPTIONAL) Boolean value indicating if the alarm already has been acknowledged.

`active` (OPTIONAL) Boolean value indicating if the state of the alarm refers to a situation in which the *alarm* actually exists.

Browse

This service is used to get a list of child items of the provided node in the address space of the server.

Message Category Call

Message Type `browse`

Payload Item object with following keys:

`id` (REQUIRED) The unique id of the parent node.

Result Item Value `null` in case of an error or a list of objects with following keys on success:

`id` (OPTIONAL) The unique id of the child node.

`name` (REQUIRED) Name of the child node.

`title` (OPTIONAL) Title of the child node.

`description` (OPTIONAL) Description of the child node.

`type` (OPTIONAL) Type of the child node.

Handle Alarm

This service is used to inform the server about an action of the user relating to an *alarm*.

Message Category Call

Message Type `handlealarm`

Payload Item object with following keys:

> **`token` (REQUIRED)** The unique `token` of an *alarm*, usually received via `subscribealarm` *publish items*.
>
> **`acknowledged` (OPTIONAL)** If set to `true` the alarm should be acknowledged.
>
> Further specifications can define additional properties, which define how the server should deal with an *alarm*. The server must return `false` if an unknown property has been received.

Result Item Value The boolean value `true` if successful, `false` otherwise.

Subscribe Data

This service is used to subscribe the value of *dataitem* at the server.

Message Category SubscribeSingleValue

Message Type `subscribedata`

Payload Item object with following keys:

> **`id` (REQUIRED)** The unique id of the *dataitem*.

Subscription Item object with same keys as the `readdata` result item:

> **`status` (OPTIONAL)** Information about the quality of the value. A missing `status` key or a falsy (e.g. `false` or `0`) value indicates a *Good* state.
>
> **`timestamp` (OPTIONAL)** Number of milliseconds since 1970-01-01 when the value or its state changed the last time.
>
> **`value` (REQUIRED)** The actual value of the data item.

Subscribe Alarm

This service is used to subscribe *alarms* via a filter at the server.

Message Category SubscribeMultipleValues

Message Type `subscribealarm`

Payload Item object with following keys:

- **`id` (REQUIRED)** The unique id of the *eventnotifier*. An empty value indicates the root of the server and therefore includes all *alarms*.
- **`filter` (OPTIONAL)** An object describing a filter which the *alarms* needs to match to be sent via a publish message.

Subscription Item object with following keys:

- **`key` (REQUIRED)** As described in JSON-CAPS.
- **`retain` (OPTIONAL)** As described in JSON-CAPS.
- **`token` (OPTIONAL)** A unique and opaque identifier for usage with `handlealarm` calls.
- **`id` (OPTIONAL)** The unique id of the *dataitem*, which is responsible for the *alarm*.
- **`timestamp` (REQUIRED)** Number of milliseconds since 1970-01-01 when the value or its state changed.
- **`priority` (OPTIONAL)** The priority of the alarm.
- **`message` (OPTIONAL)** A textual description of an *alarm* to be displayed in the UI of the HMI.
- **`acknowledged` (OPTIONAL)** Boolean value indicating if the alarm already has been acknowledged.
- **`active` (OPTIONAL)** Boolean value indicating if the state of the alarm refers to a situation in which the *alarm* actually exists.

5.4 History Periods

All history requests support the same parameters for declaring the requested time period. This section defines how the server calculates the effective period out of these parameters.

To specify a period at least two parameters out of `starttime`, `endtime` and `maxresults` must be set. If none or only one of the parameters is set, then the corresponding request must fail.

If `starttime` and `maxresult` are set, a maximum amount of `maxresult` values starting at `starttime` (including) should be returned.

If `endtime` and `maxresult` are set, a maximum amount of `maxresult` of the latest values predating `endtime` should be returned.

If `starttime` and `endtime` are set, all values which have a timestamp in the range between `starttime` (including) and `endtime` (excluding) should be returned. Optionally, the parameter `maxresult` can be used to limit the amount of returned values, starting with the earliest value.

5.5 Aggregations

In the following, various algorithms for aggregating a list of values are documented. They are mainly used for working with historic values.

If `aggreation` is used with an `interval`, the aggregation will be applied to subranges, as long as specified via `interval`.

`average`

All values get summed up first and then the total gets divided by number of numbers in the list.

`timeaverage`

All values get multiplied with their duration of occurrence and then summed up. The total will then be divided by the length of the full time period.

`minimum`

The lowest value in the interval will be returned.

`maximum`

The highest value in the interval will be returned.

`start`

The first value in the interval will be returned.

`end`

The last value in the interval will be returned.

`startbound`

The value at the beginning of the interval will be returned.

`endbound`

The value at the end of the interval will be returned.

CHAPTER 6

Mapping between OPC UA and WPCP

Since OPC UA is a broadly adopted standard in the industrial automation, it is important that WPCP provides high interoperability with it. This chapter defines how a WPCP server should behave compared to an OPC UA server. Vendors implementing WPCP servers should consider this mapping when there is already an OPC UA server for the same target platform.

The definitions allow the implementation of gateways between OPC UA and WPCP.

6.1 Address Space

In OPC UA, every node has a unique *NodeId*, which is a structure out of a *NamespaceIndex* and an *Identifier*. To map this structure to a unique string, the XML encoding for NodeIds as defined in OPC UA will be used. For example, a node in namespace 3 with the string identifier "ExampleNode" would be encoded as `ns3;s=ExampleNode`.

For resolving the relative addressing schema defined by WPCP, the OPC UA *TranslateBrowseNameToNodeId* has to be used. For the relative addressing OPC UA uses *QualifiedNames*, which consist of a *NamespaceIndex* and a *Name*. They will be encoded by converting the *NamespaceIndex* to a string and concatenating it with a semicolon and the *Name*. A *QualifiedName* in namespace 5 with the name "test" would be encoded as `5;test`.

6.2 Data Access

Read Data

The `readdata` call will be mapped to the *Read* service defined by OPC UA. The `id` property in the request payload will be mapped to the *NodeId* element in the *ReadValueId* structure. The *AttributeId* is always set to *Value* attribute, since all other node attributes are mainly used for getting information out of the data model and do not provide relevant information for displaying data.

The OPC UA response will be returned by mapping *Value* into the `value` attribute, by converting the *DataValue* to the corresponding CBOR value. The *SourceTimestamp* will be used for `timestamp`, while the *ServerTimestamp* will be ignored. The *StatusCode* will be stored as unsigned integer into the `status` property.

Write Data

To write data to the OPC UA server via `writedata` the *Write* service is used. The `id` property is mapped to the corresponding *NodeId*. Similar to the *Read* service, the *AttributeId* is always set to *Value*. The `value` sent by the WPCP client must be converted to the correct datatype before invoking the *Write* service, which could require retrieving the actual type of the node in advance.

Subscribe Data

The WPCP properties are mapped to *MonitoredItems*, monitoring the *Value* of the node specified with the `id` parameter in the same way as with the *Read* service. The additional options provided by OPC UA should be kept at their default values, or made configureable at the WPCP server.

6.3 Alarms

Subscribe Alarm

The `subscribealarm` call will be mapped to a *MonitoredItem*, monitoring the *EventNotifier* of the node specified with the `id` parameter. Mapping between OPC UA and WPCP must be done depending on the elements of an event notification.

Handle Alarm

In comparison to OPC UA, WPCP defines only one method processing the alarm. Depending on the properties set in the `handlealarm` request, different services in OPC UA must be invoked. If the `acknowledge` property is set to `true`, the *Acknowledge* method at the *AcknowledgeableConditionType* must be executed.

6.4 Historical Access

The methods in WPCP for accessing historical data can be mapped to the *HistoryRead* service of OPC UA. Depending on the properties passed to the `readhistorydata` call, the *ReadProcessedDetails* or *ReadRawModifiedDetails* struct must be used. For the `readhistoryalarm` call, the *ReadEventDetails* struct must be used.

CHAPTER 7

Implementation

To prove the usability of WPCP, a full implementation of all components with different backends has been created. The following sections describe those and give a short overview about their functionality.

7.1 wpcp.js

It is the main library for inclusion into the browser and provides an easy to use API to the user. It uses the native implementation of the WebSocket protocol of the browser for the basic communication and builds on `cbor.js` for the encoding/decoding of the CBOR content in the WebSocket messages.

The code can be found at `https://github.com/WebProcessControl/wpcp-js`.

7.2 WPCP GUI

Built on top of `wpcp.js` and the Dojo library [10], this webpage provides a simple user interface to access all functions provided by a WPCP server. It provides a simple treeview to browse the address space of the server, while all datapoints can be read, written and subscribed. Alarms provided by the server can be listed and acknowledged, too.

The code can be found at `https://github.com/WebProcessControl/wpcp-gui`.

7.3 wpcproxy

This application written in node.js acts as a full caching proxy. It allows servers to register via the `rwpcp` WebSocket subprotocol at the proxy. After a successful registration, clients are able to connect via the `wpcp` WebSocket subprotocol and can exchange data with the original server.

The code can be found at `https://github.com/WebProcessControl/wpcproxy`.

7.4 libwpcp

This subproject provides an ANSI C SDK for implementing WPCP servers. Applications using it can register callbacks, which will be executed when events like a `readdata` request from a client have been detected. The SDK ships with a backend for usage with `libwebsocket`, but can be used with any other WebSocket implementation, too. The shipped backend accepts client connections via the `rwpcp` WebSocket subprotocol and is able to connect to a proxy via the `wpcp` WebSocket subprotocol.

The code can be found at `https://github.com/WebProcessControl/libwpcp`.

7.5 WPCP2ADS

Automation Device Specification (ADS) [12] describes an interface that can be used to directly exchange data with a Beckhoff PLC. WPCP2ADS uses `libwpcp` and maps the functionality of WPCP to the corresponding ADS API calls. Since Beckhoff PLCs allow the direct execution of applications, it is possible to run WPCP2ADS directly on the PLC. This makes any additional hardware required for translating WPCP to ADS obsolete and reduces the energy and maintenance costs.

The code can be found at `https://github.com/WebProcessControl/wpcp2ads`.

7.6 WPCP2OPCUA

This executable can connect to an OPC UA server as an OPC UA client and serves as a WPCP server using the mapping defined in Chapter 6. Similar to WPCP2ADS it uses the `libwpcp` SDK, which allows the connection to a proxy using the `rwpcp` WebSocket subprotocol, too.

For connections to OPC UA the API of the offical ANSI C Stack of the OPC Foundation is used. Since there is no compatible free open source implementation available at the moment of writing this work, it is required to get a version of the stack first before compiling the application. For example the free evaluation version of the *ANSI C based OPC UA Client/Server SDK Bundle* from Unified Automation [13] contains a version of the OPC UA stack which can be used with WPCP2OPCUA.

The code can be found at `https://github.com/WebProcessControl/wpcp2opcua`.

7.7 Test Arrangement

To test interaction, a test lab with all implemented components has been set up as shown in Figure 7.1. *WPCP2ADS* is running on a Beckhoff PLC together with *TwinCAT PLC Control*. The configured PLC program controls an attached signal lamp and has a switch for input. *WPCP2ADS* connects to an instance of *wpcproxy*, which allows connections from the *WPCP GUI* running inside a browser. Parallel to that, a machine is running the OPC UA demo server provided by the OPC Foundation together with *WPCP2OPCUA*, which allows direct connections from the *WPCP GUI*. In addition to that, an instance of *UA Expert* [14] is connected to the OPC UA demo server to monitor the values of the nodes.

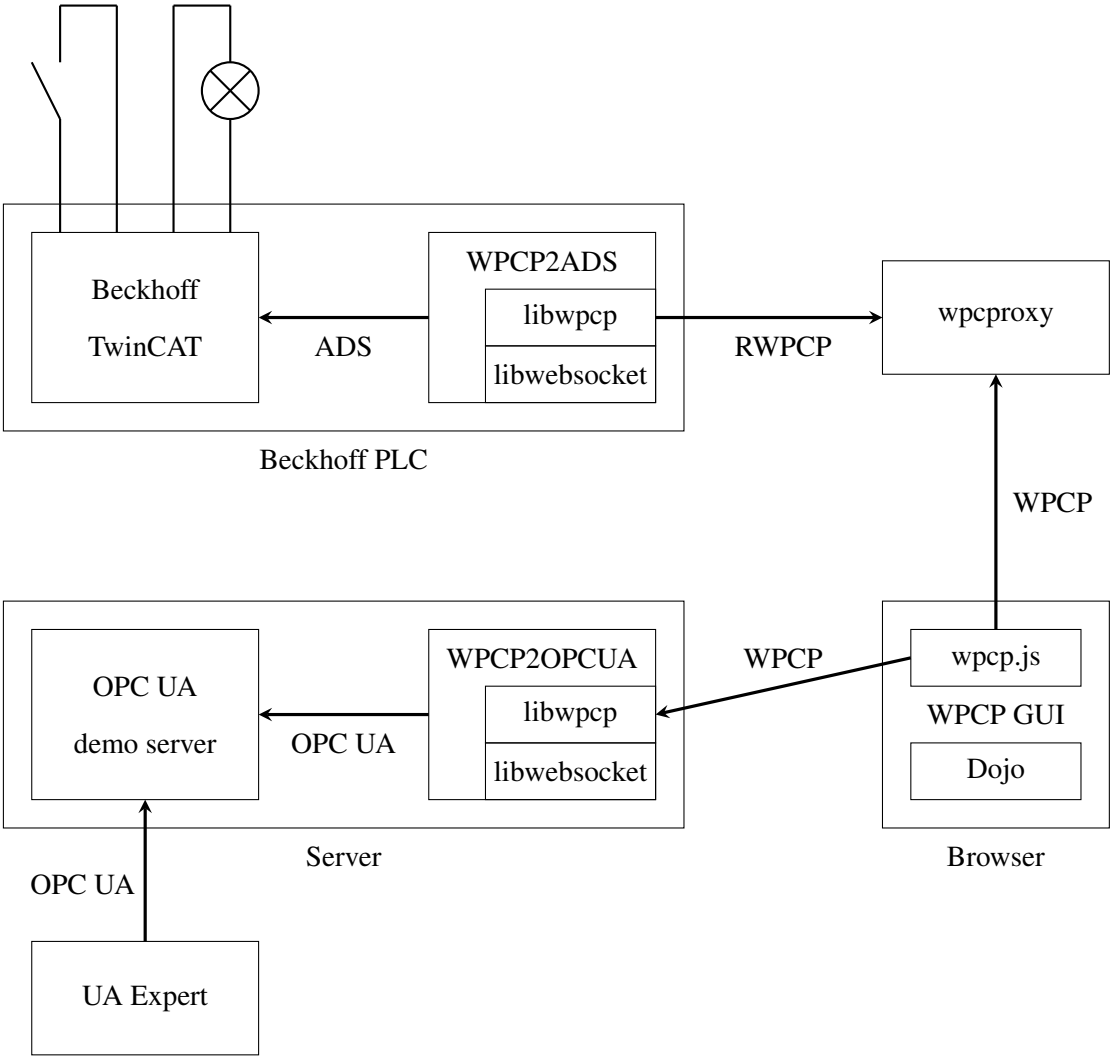

Figure 7.1: Test arrangement

The following tests have been executed to verify the implementations:

- The datapoint connected to the switch has been subscribed in *WPCP GUI*. Changing the state of the switch caused the subscribed value to change its value according to the position of the switch.
- By writing `true` or `false` to the datapoint of the lamp the light could be switched on and off.
- Browsing the OPC UA server via *WPCP GUI* showed the same tree as in *UA Expert*. Subscriptions showed the identical values for various nodes, too.

- Setting the value of a node via *WPCP GUI* has been verified by having a subscription on the same node in the *UA Expert*.
- *WPCP GUI* was also able to show the same list of alarms like the *UA Expert*, when subscribed to the same *EventNotifier*. Acknowledging it via *WPCP GUI* resulted in the correct state in the OPC UA server.

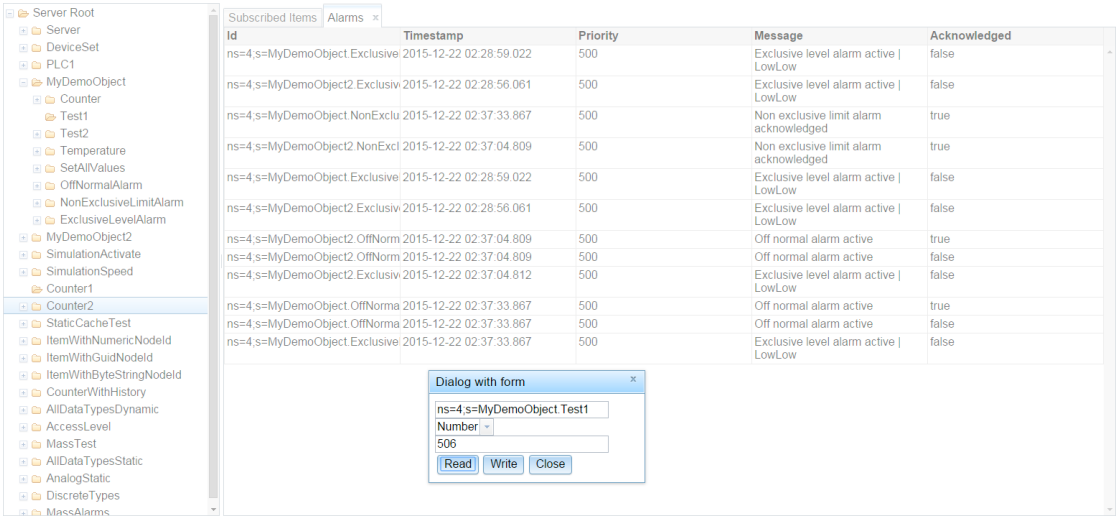

Figure 7.2: WPCP GUI connected to an OPC UA demo server

CHAPTER 8

Conclusion and Outlook

In this work, a WebSocket subprotocol for the industrial automation has been presented. Implementations for Web browsers and different kinds of data sources have proved that the protocol is able to exchange all relevant data between peers. An additional implementation of a proxy showed that is is possible to overcome most of the usual firewall restrictions. Running a WPCP server behind NAT does not require a firewall configuration, which reduces the complexity of setting up a WPCP server.

By providing a mapping to the industry standard OPC UA it has been shown that WPCP is able to fulfill usual communication requirements found in that industry.

Since WPCP defines only a communication protocol, an engineer who wants to create a visualization must build most of the parts manually. This is a complicated and error prone task.

A follow-up work should create a Web framework for building visualization for industrial automation. That framework can build upon the introduced concepts of WPCP and could provide an easy way to use this technology.

Bibliography

[1] Andrew Banks and Rahul Gupta. MQTT Version 3.1.1, October 2014.

[2] M. Belshe, R. Peon, and M. Thomson. Hypertext Transfer Protocol Version 2 (HTTP/2). RFC 7540 (Proposed Standard), May 2015.

[3] C. Bormann and P. Hoffman. Concise Binary Object Representation (CBOR). RFC 7049 (Proposed Standard), October 2013.

[4] T. Bray. The JavaScript Object Notation (JSON) Data Interchange Format. RFC 7159 (Proposed Standard), March 2014.

[5] Edward Curry. Message-oriented middleware. *Middleware for communications*, pages 1–28, 2004.

[6] T. Dierks and E. Rescorla. The Transport Layer Security (TLS) Protocol Version 1.2. RFC 5246 (Proposed Standard), August 2008. Updated by RFCs 5746, 5878, 6176.

[7] I. Fette and A. Melnikov. The WebSocket Protocol. RFC 6455 (Proposed Standard), December 2011.

[8] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee. Hypertext Transfer Protocol – HTTP/1.1. RFC 2616 (Draft Standard), June 1999. Updated by RFCs 2817, 5785, 6266, 6585.

[9] Roy Thomas Fielding. *Architectural Styles and the Design of Network-based Software Architectures*. PhD thesis, University of California, Irvine, 2000.

[10] The Dojo Foundation. Dojo Toolkit, 2015. https://dojotoolkit.org/.

[11] P. Gansterer. webMI Implementation Guideline, August 2008. http://www.atvise.com/.

[12] Beckhoff Automation GmbH. TwinCAT ADS, 2015. http://infosys.beckhoff.com/.

[13] Unified Automation GmbH. ANSI C based OPC UA Client/Server SDK Bundle, 2015. https://www.unified-automation.com/products/server-sdk/ansi-c-ua-server-sdk.html.

[14] Unified Automation GmbH. UaExpert, 2015. https://www.unified-automation.com/products/development-tools/uaexpert.html.

[15] K. Hartke. Observing Resources in the Constrained Application Protocol (CoAP). RFC 7641 (Proposed Standard), September 2015.

[16] S. Hennig, A. Braune, and M. Damm. JasUA: A JavaScript Stack enabling Web browsers to support OPC Unified Architecture's Binary mapping natively. In *Emerging Technologies and Factory Automation (ETFA), 2010 IEEE Conference on*, pages 1–4, Sept 2010.

[17] Ian Hickson. The WebSocket API. Candidate recommendation, W3C, September 2012. http://www.w3.org/TR/2012/CR-websockets-20120920/.

[18] ISO/IEC. IEC 62541: OPC Unified Architecture, 2008.

[19] ISO/IEC. IEC 19464: Advanced Message Queuing Protocol (AMQP) v1.0 specification, 2014.

[20] G. Kaplan. Ethernet's winning ways. *Spectrum, IEEE*, 38(1):113–115, Jan 2001.

[21] Frank Matthias Kovatsch. *Scalable Web Technology for the Internet of Things*. PhD thesis, Diss., Eidgenössische Technische Hochschule ETH Zürich, Nr. 22398, 2015.

[22] Nurzhan Nurseitov, Michael Paulson, Randall Reynolds, and Clemente Izurieta. Comparison of JSON and XML Data Interchange Formats: A Case Study. *Caine*, 9:157–162, 2009.

[23] T. Oberstein. The Web Application Messaging Protocol, 2014. http://wamp.ws/spec/.

[24] OPC Foundation. https://opcfoundation.org/news/press-releases/2134/. Accessed: 2015-08-31.

[25] Tatu Paronen. A web-based monitoring system for the Industrial Internet. Master's thesis, Aalto University, April 2015.

[26] Z. Shelby, K. Hartke, and C. Bormann. The Constrained Application Protocol (CoAP). RFC 7252 (Proposed Standard), June 2014.

[27] P. Srisuresh and K. Egevang. Traditional IP Network Address Translator (Traditional NAT). RFC 3022 (Informational), January 2001.

[28] Hallvord Steen, Julian Aubourg, Jungkee Song, and Anne van Kesteren. XMLHttpRequest level 1. W3C working draft, W3C, January 2014. http://www.w3.org/TR/2014/WD-XMLHttpRequest-20140130/.

[29] Anne van Kesteren. Cross-origin resource sharing. W3C recommendation, W3C, January 2014. http://www.w3.org/TR/2014/REC-cors-20140116/.