RESTful Industrial Communication with OPC UA

Article *in* IEEE Transactions on Industrial Informatics · October 2016

DOI: 10.1109/TII.2016.2530404

CITATIONS
54

READS
5,198

3 authors, including:

Sten Grüner
ABB
37 PUBLICATIONS **243** CITATIONS
SEE PROFILE

Julius Pfrommer
Fraunhofer Institute of Optronics, System Technologies and Image Exploitation IOSB
32 PUBLICATIONS **604** CITATIONS
SEE PROFILE

RESTful Industrial Communication with OPC UA

Sten Grüner, Julius Pfrommer, and Florian Palm

***Abstract*—Representational State Transfer (REST) is a widespread architecture style for decentralized applications. REST proposes the use of a fixed set of service interfaces to transfer heterogeneous resource representations instead of defining custom interfaces for individual applications. This paper explores the advantages of RESTful architectures, i.e., service-oriented software architectures comprised of RESTful services, in industrial settings. These include a) communication advantages, such as reduced communication overhead and the possibility to introduce caching layers, and b) system design advantages including stable service interfaces across applications and the use of resource-oriented information models in cyber-physical systems. Additionally, a RESTful extension to the OPC Unified Architecture (OPC UA) binary protocol is proposed in order to leverage these advantages. It requires only minimal modifications to the existing OPC UA stacks and is fully backwards-compatible with the standard protocol. Performance benchmarks on industrial hardware show a throughput increase up to a factor of eight for short-lived interactions. This reduction of overhead is especially relevant for the use of OPC UA in the emerging Industrial Internet of Things.**

I. Introduction

Currently, the traditional manufacturing industries and the information and communications technology industry are coming together in the emerging Industrial Internet of Things (IIoT) [1]. Both lineages bring along their favored communication protocols. On the one hand, this includes protocols designed for use on the Internet, such as Message Queue Telemetry Transport (MQTT) and the Constrained Application Protocol (CoAP). These are designed to operate in settings where communication is generally short-lived and comes without Quality-of-Service guarantees. On the other hand, the protocols designed for the manufacturing and automation industry, such as fieldbuses, real-time Ethernet protocols [2] and OLE for Process Control (OPC Classic) assume long-lived connections in confined and tightly controlled environments. The more recent OPC Unified Architecture (OPC UA) combines features of both worlds and is a major contender for the application protocol which will be used in Ethernet-based communication in the domain of cyber-physical systems.

RESTful communication is a set of design principles that underlies the hypertext-based web and is commonly used for the interfaces of decentralized software architectures. These design principles are technology-independent and are not bound to a specific protocol.

S. Grüner and F. Palm are with the Chair of Process Control Engineering, RWTH Aaachen University, Aachen, 52064, Germany (e-mail: s.gruener@plt.rwth-aachen.de, f.palm@plt.rwth-aachen.de).

J. Pfrommer is with Fraunhofer Institute of Optronics, System Technologies and Image Exploitation (IOSB), Karlsruhe, 76131, Germany (e-mail: julius.pfrommer@iosb.fraunhofer.de).

In this paper, we discuss the advantages of RESTful interfaces for industrial communication and propose a lightweight set of extensions to make OPC UA RESTful. These follow the same objective as the CoAP transfer protocol, i.e., the creation of a lightweight RESTful interface. However, instead of changing to an entirely different protocol, existing OPC UA based applications can profit from the advantages of RESTful communication with minimal changes to their communication stacks.

This publication is an extended version of our previous work published in [3]. The paper is organized in the following way: It begins with an overview on related work in Section II. After presenting some background on OPC UA and RESTful communication, the advantages of RESTful service architectures for industrial applications are discussed in Section III. The main contributions of the paper are as follows:

- In Section IV, the missing properties of OPC UA that are required to realize RESTful services are identified. Subsequently, performance advantages of REST and possibilities of using stateless application and transport layers are illustrated with the help of industrial application use cases.
- In Section V, a set of extensions to OPC UA is proposed to fill this gap for a subset of OPC UA services.
- The resulting performance improvements are evaluated on industrial hardware in Section VI.

Finally, a summary and an outlook towards future work are provided in Section VII.

II. Related Work

A. Service-Orientation in the Industry

Several authors have proposed service-oriented architectures (SOA) for distributed industrial applications.

Jammes and Smit [4] describe opportunities and challenges of SOA for industrial automation and especially for the manufacturing industry. They consider a SOA implementation using web services at the device level and present a set of advantages of SOA, most notably, uncoupling of physical and logical aspects of the system, scalable service composability and a simplified application development.

De Sousa et al. [5] describe the use of a service-oriented middleware to connect the shopfloor with the management Enterprise Resource Planning (ERP) level.

A number of successful SOA applications in different industrial branches were presented in [6] including manufacturing and logistics, train car management systems and semiconductor processing equipment.

In [7], the applicability of SOA as a programming paradigm for industrial robotic cells has been validated. The authors conclude that SOA supports automation engineers in focusing

on the area of one's expertise and to reduce the interfacing overheads between different components of a cell. In this investigation, programming that used a SOA framework has been less time-consuming and hence more cost-effective than traditional object-oriented techniques.

The question of whether SOA can be scaled down to the industrial device-level has been studied in [8]. In this work, synergies between OPC UA and Devices Profile for Web Services (DPWS) where OPC UA is used for server-client and DPWS for peer-to-peer communication are presented.

The OPC UA binary protocol has been compared with SOAP [9] in terms of performance with a focus on subscription mechanisms and security models of the standardized protocol.

We are not aware of prior work that explicitly discusses RESTful services in industrial applications.

B. REST for Constrained Devices

The idea of using RESTful services to interact with resource-constrained physical devices was proposed over 10 years ago [10] and is frequently proposed in the context of Internet of Things architectures [11].

CoAP [12] is a transfer protocol that has been developed especially for constrained devices that are used in the context of IoT. CoAP has been developed by the IETF Constrained RESTful Environments working group and is positioned as a lightweight binary alternative for the text-based Hypertext Transfer Protocol (HTTP) which is primarily used to implement RESTful architectures. Another difference between CoAP and HTTP is that the former runs over UDP while the latter is a TCP-based protocol. CoAP also defines some mechanisms that are not included in HTTP, e.g., subscriptions.

III. An Introduction to REST and OPC UA

A. Representational State Transfer

REST is an architecture style for resource-oriented distributed applications [13], [14]. It proposes the use of simple, uniform interfaces for the exchange of resource representations. A resource can be a physical or a virtual entity. The interface does not depend on the resource (e.g., only CRUD (create, read, update, delete) operations are available), whereas the representations of the different resources are heterogeneous.

Today, RESTful HTTP is used as a simple way to implement web services where the Uniform Resource Locator (URL) denotes a resource, e.g., http://company.com/site/plant/line/mill42/toolingstatus. Sometimes, arguments are added to the URL to emulate a remote procedure call. But that is not entirely compliant with the REST approach, as it adds a custom API definition on top of RESTful HTTP [13]. This API needs to be communicated out-of-band, which goes contrary to the requirement for uniform interfaces used to exchange resource representations. The solution of the hypertext-based web for interacting with unknown resources is the exchange of HTML websites with embedded JavaScript (a well-known resource representation format) that internally contains the facilities to interact with non-standard resource descriptions.

Richardson and Ruby [14] describe four properties of RESTful web services as a part of a resource-oriented architecture:

1) Addressability: Resources are assigned to an address that acts as their identifier and that contains enough information to establish a communication channel.

2) Statelessness: The requests and responses must be self-contained and do not rely on the receiver storing information per each connection. This requirement may be weakened to allow a transport medium with support for packet ordering, retransmission, as well as authentication and encrypted communication. Statelessness enables asynchronous communication where multiple requests are sent in parallel without waiting for a response. Moreover, it reduces the requirements on the main memory for servers that need to handle many connections.

3) Connectedness: Connectedness describes how resources reference each other in their representations via their identifying addresses. Users can then browse these links and find out about the context and relationships of the resource.

4) Uniform Interface: The interface for interacting with resource representations does not change between resources. In HTTP, the protocol used to access websites and the available operations (GET, POST, PUT, DELETE and some more) are the same for every website and type of transferred media. In service-oriented approaches, many services might be available at one location, but every service has different arguments and return values that users need to consider in their applications.

There exist a number of protocol bridge implementations that expose a different service paradigm via RESTful services, for example the bridge between MQTT and RESTful HTTP protocols [15] or the commercial tool [16] acting as a gateway between OPC UA servers and HTTP clients by non-standard mapping.

B. OPC Unified Architecture

OPC UA is a protocol for industrial communication and has been standardized in IEC 62541. Developed as the successor to the widely used OPC Classic, it has become a major contender for flexible communication in industrial applications without hard real-time requirements. The OPC Foundation drives the continuous improvement of the standard, the development of companion specifications to incorporate existing information models and the adoption of OPC UA in the industry by hosting events and providing the infrastructure and tools for compliance certification. Existing companion specifications in the manufacturing domain are for example FDI, ISA-95, IEC 61131-3 and MTConnect.

OPC UA is a client-server communication protocol. The server provides access to data and functions that are structured in an object-oriented information model with which clients can interact via a set of standardized services as listed in Fig. 1. A pair of request-response data structures is defined for each service. These structures are built up from basic types such as integers etc. and encoded either in a custom binary format or in XML. The simplified structures of a *ReadRequest* and the corresponding *ReadResponse* packet are shown in Fig. 2a and 2b, respectively. Most services require the client to establish a secure channel and a session prior to using them.

The information model on which the services operate combines ideas from object-orientation and semantic technologies

Discovery Service Set
- FindServers*
- GetEndpoints*
- RegisterServer*

SecureChannel Service Set
- OpenSecureChannel
- CloseSecureChannel

Session Service Set
- CreateSession
- ActivateSession
- CloseSession
- Cancel

NodeManagement Service Set
- AddNodes*
- AddReferences*
- DeleteNodes*
- DeleteReferences*

View Service Set
- Browse*
- BrowseNext
- TranslateBrowsePathsToNodeIds*
- RegisterNodes
- UnregisterNodes

Query Service Set
- QueryFirst*
- QueryNext

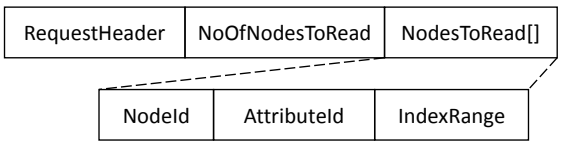
Attribute Service Set
- Read*
- HistoryRead
- Write*
- HistoryUpdate

Method Service Set
- Call*

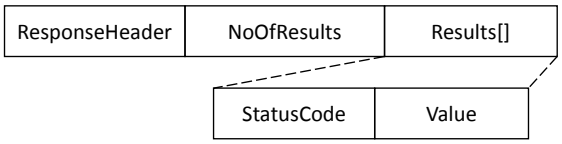
MonitoredItem Service Set
- CreateMonitoredItems
- ModifyMonitoredItems
- SetMonitoringMode
- SetTriggering
- DeleteMonitoredItems

Subscription Service Set
- CreateSubscription
- ModifySubscription
- SetPublishingMode
- Publish
- Republish
- TransferSubscriptions
- DeleteSubscriptions

Fig. 1: Services defined in the OPC UA standard. Services that are inherently stateless are marked with a star.



(a) ReadRequest message structure.



(b) ReadResponse message structure.

Fig. 2: The request and response messages of the *Read* service. It is used to read attributes of nodes in an OPC UA information model. The *IndexRange* field is used to denote subsets of attribute values with large (multidimensional) arrays.

(ontologies and their triple-representation). At its core, an OPC UA information model is a graph consisting of nodes and references of the form (`source-node`, `referencetype`, `target-node`), linking two nodes at a time. Every node is of one of the eight node types listed in Table I and may represent a variable, a method, a (custom) data type and so on. The node type defines the attributes the node contains. Some attributes, such as the unique node identifier and a readable name, are common to all nodes. The references between nodes are tagged with a reference type which gives meaning to the relationship. For example, a *VariableNode* has a *HasTypeDefinition* reference to the node representing the data type of the variable. Some reference types are defined to be hierarchical and may not form directed cycles. One example of this is an *ObjectNode* which has a hierarchical reference to the variables and methods it contains. These hierarchical relations and the nodes' name can be used to identify nodes via a browse path such as "/Objects/Plant1/Cell5/Lathe2/Temperature", which the server translates into matching node identifiers.

TABLE I: Node types in the OPC UA information model.

Type Name	Usage
DataTypeNode	Define simple and structured data types
VariableNode	Represent a value (scalar, array, multi-dimensional array)
VariableTypeNode	Define requirements for VariableNodes (value type, array size and so on)
MethodNode	Define callable remote procedures
ObjectNode	Represent an object made up of variables, methods and further objects
ObjectTypeNode	Define requirements for ObjectNodes (contained variables, methods and so on)
ReferenceTypeNode	Define a type for references between nodes (hierarchical or non-hierarchical)
ViewNode	Provide access to a subset of nodes

OPC UA allows its information model to be fully introspected and—in theory—modified at runtime by the users via the standardized services. Common and reusable (for some domains of interest) information models are released in so-called companion specifications to enable interoperability across solution providers. These standards are developed jointly with the standardization bodies of the specific domain and industry working groups. In many cases, they provide a mapping of established standards onto the OPC UA information model.

C. Advantages of RESTful Services in Industrial Settings

Resource-Oriented Information Model: Many service-oriented architectures use services as a façade to abstract away replaceable service providers for a loose coupling of components. In cyber-physical systems, however, operations are never fully detached from some physical counterpart. Additionally, system components are not just service providers, but have an inherent state. Due to this fact, interfaces for interacting with resource representations are very idiosyncratic. Furthermore, if the resource-oriented information model can be introspected, it is possible to explore the structure of resources and their interdependencies according to the REST principles of connectedness.

Stable Interfaces: There are only few services in a RESTful architecture, because the differences of the resources are encapsulated in the representation formats. Since they are intended for general purpose, RESTful service definitions are very stable. When introspection capabilities are part of the RESTful services, components are likely to remain usable even over the long-term future. As long as the RESTful services are accessible, no further out-of-band information is necessary to connect and discover the representation formats that are used.

Reduced Resource Consumption: For short-lived interactions, stateless interfaces reduce the overhead of session initialization and termination. Furthermore, the server only needs to store minimal per-connection information, such as a socket descriptor. Since no per-connection state is managed, concurrent message processing on multiple processor cores requires less synchronization points.

Caching and Load Balancing: The load on a constrained server can be reduced by caching and load balancing. In both cases, user requests are transparently routed through an intermediate layer on the network level. This is greatly simplified if the routing does not have to consider a mapping of sessions to back-end servers. By load balancing we mean distributing the workload from a particular server to multiple caches or further servers between it and its clients. For caching, resource representations that were recently sent out are *replayed* until they are invalidated by the original source. This advantage is addressed in an OPC UA-specific use-case in Section IV-B4.

IV. RESTful OPC UA

A. How RESTful is OPC UA already?

In the following section, we compare the requirements for RESTful services (cf. Section III-A) with the specification of OPC UA:

1) Addressability: Addressability is provided by the protocol. All of the nodes have a unique identifier in the server's address space and can be reached via their browse path over the hierarchical relation to their ancestor nodes.

2) Statelessness: In general, OPC UA cannot be considered stateless. First of all, the protocol defines compulsory secure channels and sessions that will expire if they are not renewed regularly. Secondly, many of the OPC UA services are inherently stateful. An example for such a service is the continuation of a browse request where subsequent requests return the results that could not fit into the preceding responses. Also, every operation regarding subscriptions relies on session-based states in the server.

3) Connectedness: It is a core concept of OPC UA to have nodes that reference each other. References to external servers are accomplished with expanded node identifiers that contain an optional server URL.

4) Uniform Interface: The services defined in OPC UA are fixed and cannot be changed by the user. Note that the *Call* service allows arbitrary combinations of input and output arguments. Yet, there is a standard way to discover this signature in the information model via the corresponding *MethodNode*.

This evaluation shows that statelessness is the only missing RESTful feature of OPC UA. The idea of stateless OPC UA communication over TCP seems to be controversial at first glance since communication over TCP implies a certain state – the established socket. Even though RESTful interfaces are usually implemented by using HTTP and TCP, a stateful transmission is not required by REST definition.

By saying "RESTful OPC UA", we therefore mean a stateless condition on the application layer in the first place. Being stateful in terms of OPC UA refers to mechanisms of message chunking, which is used to split long OPC UA messages into several smaller packets (chunks) and establishing of secure channels and sessions.

TABLE II: Possible feature profiles concerning stateless communication on the application layer (AL, here the OPC UA stack) and the transport layer (TL).

	Stateful AL	Stateless AL
Stateful TL (TCP)	Standard	Stateless Interactions with Retransmission
Stateless TL (UDP)	Security & Sessions w/o Retransmission	Resource Efficient

The four combinations of different modes for the application and the transport layer are presented in Table II. The left column includes use cases where security and session-related features, e.g., subscriptions of OPC UA are required by the application. Using a stateless network transport (the lower left cell) limits the retransmissions of OPC UA packets. The consequences are described in Section V-C in more detail. Stateless application layer, i.e., RESTful OPC UA use cases in the right column, are best suited for effective caching and load balancing as well as for the efficient handling of short-lived interactions that are presented in the next section. The question of how much performance improvement is achievable with RESTful OPC UA over UDP (the lower right cell) is addressed in Section VI.

B. Use-Cases for RESTful OPC UA

1) Short-lived Communication Use-Case: Scenarios with constrained or non-scalable servers aim for situations where the server could not normally cope with a high load of many clients/many requests. Popular services on the Internet experience loads of up to a million connected clients on a single device. This poses considerable constraints on the memory, which can be stored per-connection in RAM, and the complexity of the involved data structures. With stateless connections, the server only has to keep a socket connection data without any additional per-client information.

We call a communication setting short-lived if the following criteria are fulfilled:

- there are many clients communicating with the server,
- a single interaction with a client is relatively short-lived,
- there are high delays between interactions of a particular client (higher than the lifetime of the session-related data).

Stock monitoring is an example of highly short-lived communication. Modern high-bay storages contain tens of thousands of pallets. The periodic status transmission of the pallet status (e.g., its temperature and position) can be handled with existing server applications. However, peak loads for alert notifications, for example 30,000 pallets trying to send status information when the temperature drops under a minimum level, could easily bring conventional systems to their limits. Here, the limiting quantity is not only the required network throughput, but also the ability of the server and the underlying operating system to manage concurrent stateful access under the given time constraints.

Memory limitations are especially important in case of short-lived interactions with the server. These problems emerge in the simplest OPC UA server profile, namely the nano profile that is used on many constrained devices. The profile prescribes at least one session that shall be implemented on the server. In the worst case (when only a single session per server is supported), it means that the server does not accept new connections until the session expires if a client does not close the session properly.

2) Low-Energy Clients: Low-energy clients, for example, distributed IoT devices running on batteries, must reduce the communication overhead to increase their lifetime. In this setting, RESTful OPC UA together with a session-free transport protocol like UDP greatly reduces the number of round trips necessary to exchange basic information.

It is even possible to have clients wake up, send a write request that updates a data value to the server and go back to sleep without waiting for a response. An example for those types of clients are low-energy Bluetooth devices for which OPC UA interfaces have already been studied [17].

3) RESTful OPC UA over Hybrid Fieldbuses: In recent years, hybrid bus protocols (sometimes known as industrial Ethernet) [2] have been proposed where both a deterministic transportation mode (e.g., based on assigning time slices in a master/slave setting) and nondeterministic communication via Ethernet are combined. Communication partners can establish deterministic communication at runtime as long as it is supported by intermediate routing equipment.

Here, RESTful OPC UA can be used for real-time communication where bandwidth is inherently limited. For reads/writes to predefined nodes, the message size is fixed or an upper bound exists. Also, the request/response pairs are the only exchanged messages. Since the communication is stateless, no session or secure channel keepalive is necessary.

Short-lived data exchanges are suitable for the non-realtime window of a hybrid bus, for example, for exchanging meta information about the real-time data.

4) Caching/Load Balancing on Network Level: Both, caching and client-transparent redundancy applications can be realized with OPC UA and have been, e.g., considered in [18]. Still, using RESTful OPC UA requests has a number of advantages, most notably the drastic simplification of the caching servers that are discussed in the following.

The load on a constrained server can be reduced by caching, if small delays in node updates are within the service-level's requirements. A caching/load balancing server as shown in Fig. 3, consists of a load balancing unit and a number of caches that cache server's responses.

Firstly, let us consider a server consisting of only one cache. If a request for an already requested attribute comes in, it will replay the original response as long as it has not timed out. As a remark, only idempotent service requests should be cacheable, which applies to *Read* and *Browse* services. The expiry header entry is used in the communication with the back-end OPC UA server (cf. Section V-D) if no particular expiration policy is set in the client's request. The caching server indicates the lifetime of the data to the client, while the original server indicates the lifetime to the cache. This means

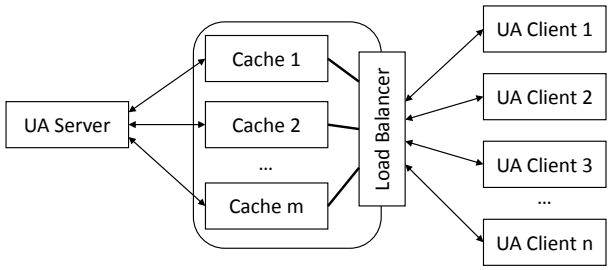


Fig. 3: A mixed caching/load balancing architecture.

that the scalability can be improved by caching infrastructure even if the client does not support caching headers (i.e., it ignores the expiration timestamp). The logic of the caching server can be simplified by using stateless requests since only a minimal amount of OPC UA semantics has to be known to the server (for example, no knowledge of sessions is required).

Additionally, multiple caches can be bundled by means of a load balancer that distributes requests to multiple caching servers, e.g., by using a round robin scheduling algorithm. Load balancing with such a simple algorithm is only possible if requests are stateless. In this case, the balancing server does not need to ensure that messages belonging to the same session are processed on the same cache.

The presented infrastructural facility, the caching/load balancing server, operating between the clients and the backend servers can be used to improve the functionality and the performance of OPC UA aggregating servers [19], [20].

V. Realizing RESTful OPC UA

To reduce the described communication overhead and enable the described scalability features, we propose a minimally invasive extensions of the OPC UA binary protocol that enables stateless REST-style communication.

A. Application Layer: Secure Channel Zero and Session Zero

Reasonable assumptions have to be implicitly made for clients that use the stateless communication. On the level of OPC UA, we disable the chunking of messages in order to keep requests self-contained and set buffer sizes to a significantly large amount for common message exchanges, e.g., 65 kB. A small cache for partially received messages is still necessary, since TCP is a byte-stream oriented protocol, i.e., it is not preserving the OPC UA message boundaries.

Firstly, we omit the need for elaborate handshakes. If the client does not initiate a HEL/ACK handshake in its first message, it will be identified as a stateless one. Even if a stateless client does not open a secure channel and subsequently create a session, it will still be able to send messages with reference to the secure channel with the identifier 0 and also to the *anonymous session* with the identifier 0 (also called "session zero"). Session zero will give the client access to the marked UA services from Fig. 1. The most important services among them are: *Read*, *Write*, *Browse*, *TranslateBrowsePathsToNodeIds*, *AddNodes*, *AddReferences*, *DeleteNodes*, *DeleteReferences* and *Call*. All these services are invoked with a self-contained

message that does not rely on previous messages and can be answered with a single, self-contained response message. If the response does not fit into a single message, the status code *Bad_ResponseTooLarge* will be returned. User access control for session zero on the node level is left as an implementation choice. For example, if manipulating the data model by an anonymous client was risky in terms of process safety, the server might block those requests completely or only allow them on a certain subset of nodes.

We were able to add the proposed changes to an open source implementation of the OPC UA stack (cf. Section VI-B) with a patch-set that changes about 50 lines of code in total. The changes do not alter the compatibility with standard-compliant clients, which can continue to access the server in parallel to stateless clients. This is possible since the standard requires the secure channel and the session IDs to be greater than 0. Therefore, standard compliant stateful clients will never initiate a stateless service call and both, the stateful and the stateless requests can simply be distinguished by the server. Conversely, REST-enabled clients can probe whether the server supports stateless requests by activating a session zero.

B. Security

Security is of paramount importance in the domain of industrial communication and one of the key features of OPC UA. One has to distinguish between three different security options when using OPC UA: no security, signed messages to ensure its origin and integrity and fully encrypted messages [18].

Public-key cryptography, which OPC UA relies on, requires a secure channel establishment, e.g., for certificate and key exchange. These mechanisms conflict with the idea of self-contained, stateless requests. Still, there exist different possibilities for a safe, stateless communication that can be implemented in the context of RESTful OPC UA. The first one is the use of out-of-band shared secrets as discussed in the example of wireless sensor networks in [21] in conjunction with constrained devices [22]. The major design decision concerns the development of an out-of-band key distribution system, e.g., whether the keys can be installed by the device-vendor. Once a key pair is available for the server and the client, the security mechanisms of OPC UA for encrypting and signing can be used with minor adoptions.

A second possibility is to shift security issues down the communication stack to some protocol that underlies OPC UA and to disable the security features of OPC UA. Examples for an encapsulation protocol that handles security are Transport Layer Security (TLS) for TCP or Datagram Transport Layer Security (DTLS) for UDP (cf. Section V-C). Clearly, these protocols will establish a stateful secure channel, while the application layer can still be kept stateless.

C. Transport Layer: OPC UA over UDP

The specification of OPC UA clearly requires the usage of TCP for all messages. However, implementations of OPC UA server and OPC UA clients can both be easily adapted to use UDP sockets. In this section, implications of UDP usage are discussed. We now assume that only stateless requests are used as introduced earlier in this section.

TCP uses retransmission of the lost packets to ensure their delivery. Packets from higher communication levels also get fragmented and reassembled by the protocol depending on the underlying levels. These mechanisms have a great impact on the predictability of the communication delays. In case of real-time communication, i.e., when data has a certain lifetime, e.g., in video streaming, packets that arrive too late are usually useless for the application. The retransmissions therefore only generate additional load on the network infrastructure. The general rule of thumb for such systems is to prefer TCP for signaling (the integrity of signaling data is important) and UDP for the actually exchanged data. The issue of lost data packets is then handled by the application itself and not by the network protocol since it has no knowledge of the application's environment and data semantics.

A significant advantage of using UDP is the decrease of interaction between the server and the client in terms of number of exchanged packets and hence time (cf. calculations in Section VI-A and evaluation in Section VI). UDP avoids head-of-line blocking making the processing of messages more efficient. Moreover, less requirements on the communication capabilities of the underlying operating system/hardware are imposed. Another advantage is a more deterministic transport delay, since no reordering or retransmission is performed.

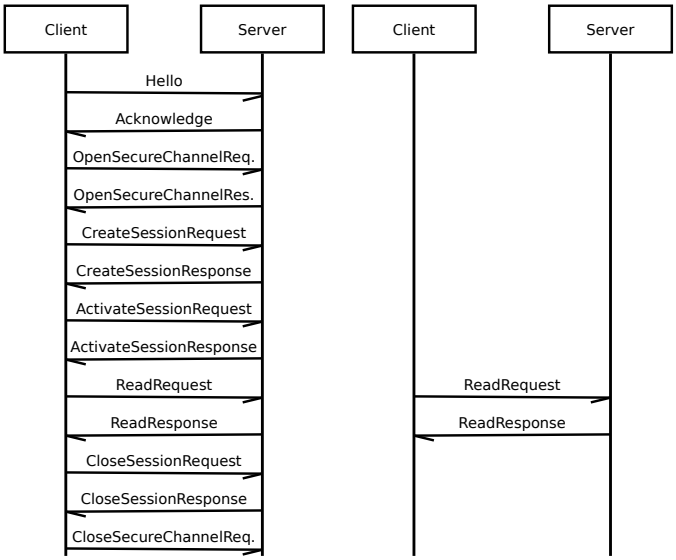
The proposed extension of standard OPC UA includes a redefinition of the used socket interface (in fact, both the TCP and UDP based transport can be used at the same time and even port on a server/client) and a minor adoption of existing APIs (that are not standardized). Our current implementation makes use of a standard request/response field called "sequenceId" to relate an outgoing request to an incoming response. This mechanism is stateless, since the server only has to copy the sequence identifier from the request to the response.

The unreliable transport puts high demands on the implementation of OPC UA message chunking mechanism, since arbitrary chunks may not arrive at their destination. One way of solving this problem is to re-implement the message retransmission mechanism on the application layer (which is done, e.g., by CoAP [12]). This strategy is realizable for OPC UA by redefining of the error messages semantics and by introducing additional headers for retransmission signaling. For the presented evaluation, the chunking was disabled completely.

D. Expiration Tags for Cacheable Responses

A further requirement of REST architecture is the caching of service responses. A response should be explicitly marked as cacheable. In this case, the response should contain a time interval that defines the lifetime of the response's data. Caching implies a time-synchronization between the server and the client. This issue is not in the scope of OPC UA and has to be handled by the underlying operating system, e.g., by means of Precision Time Protocol (PTP).

Every service response in OPC UA already includes a time stamp – the time of message creation. This data is contained in the so-called response header of every request and



(a) Standard OPC UA. (b) RESTful OPC UA.

Fig. 4: Message exchange diagrams for a single *Read* request.

is comparable with HTTP's "Date" header field. The simplest cache control strategy is to add a time stamp or time span variable to the header to define the data lifetime. There are two alternative methods for describing the time span: the first one is to take a built-in "DateTime" data type that is used for time stamps to denote the expiration of the data, while the second one is to take a "Duration" data type that describes time periods. For our prototypical implementation, we have chosen expiration timestamps in favor of duration timestamps.

The second design decision relates to requests querying for multiple nodes. Here, either one expiration time stamp for the whole request, or an array containing an expiration time for every requested data item can be specified depending on the request's context. The expiration information can be embedded into the additional header structure of the response header in a standard compliant way. Additional header information is claimed to be "reserved for further use" by the standard and is, therefore, ignored by standard clients. A use-case involving cache headers for purposes of caching and load balancing was outlined in Section IV-B4.

VI. Evaluation

In this section, we evaluate the performance improvements achieved by the proposed extensions to OPC UA.

A. Roundtrip Reduction via Stateless OPC UA

Stateless service invocation for OPC UA does not only bring along the architectural advantages of RESTful services, but also a significant performance improvement for certain scenarios.

A typical message exchange for a binary OPC UA service call is shown in Fig. 4a. The client initiates the communication with a "hello" (HEL) message that is confirmed by the server with an "acknowledge" (ACK) message. Consequently, a pair of messages is exchanged to open a secure channel and define its properties, e.g., the encryption. The next pair of messages creates a new session that holds the client's state on the server (this step is omitted if a session has been established in the past). The session activation request either activates the just created session or a past session and restores the client's context information on the server. The connection is initiated at this point. After the initiation, the actual service request can be performed. Eventually, the client has to close the session and secure channel.

There are 21 packets that need to be exchanged between the client and the server for the described single service invocation: 13 packets can be found in Fig. 4a, further 8 TCP packets like SYN are omitted. In the worst case, i.e., for a client performing a single *Read* service invocation without previously establishing a session, 11 additional packets need to be exchanged in order to perform the *Read* request. For every exchange, the corresponding round trip time (RTT) has to be taken into account. The number of exchanged packets does not depend on whether the binary or SOAP communication protocol of OPC UA specification is used. However, the usage of SOAP increases the size of exchanged data due to textual data encoding and XML markup overheads.

A message exchange needed for a stateless *Read* service invocation over TCP is presented in Fig. 4b. The amount of exchanged messages goes down from 13 to 2 (or from 21 to 10 if we count all TCP packets) which is equivalent to an improvement of more than 50%.

If the duration of the message exchanges is taken as the yardstick, the improvement depends on the communication delay between client and server and their hardware. Due to the described advantages of the stateless message exchange, a decrease of memory-consumption in both, the server and the client can be expected. For more information and quantified data, please refer to the evaluation and discussion in Section VI-C.

There is still a considerable number of TCP signaling exchanged in both the stateful and the stateless service invocation scenarios. The number of exchanged packets can therefore be further reduced by switching from TCP to UDP. In this case, there are only two packets that need to be exchanged for a complete *Read* service call as shown in Fig. 4b. As discussed in Section V-C, reducing the number of exchanged messages has a price in terms of reliability and maximal message size that can be transmitted.

B. Evaluation Setup

The open source OPC UA stack open62541 [23] is used for evaluation purposes. The stack is licensed under GNU Lesser General Public License (LGPL) with a static linking exception making it usable for both open source and commercial projects.

The open62541 project implements OPC UA servers and clients as a layered architecture which handles the connection establishment, secure channel and session management, as well as the encoding and decoding of OPC UA messages (currently only binary protocol is supported). In addition to the communication functionality, open62541 implements most of the OPC UA services and a data store for the underlying information model (nodes and references). The stack runs

on different POSIX-compliant platforms, embedded systems and Microsoft Windows. Compared to other open source and commercial stacks, open62541 is significantly modularized and has the smallest footprint in terms of binary size and lines of code. This made the integration of the proposed extensions straightforward. The extensions have now become part of the stack distribution, but need to be activated manually prior to compiling.

The RESTful extensions were evaluated on three systems with two embedded systems among them.

The first system is a WAGO-750-860 fieldbus controller. The controller has a 44 MHz ARM7TDMI CPU, 16 MB of RAM and runs μClinux with a 2.6.34 kernel. The system was chosen as a representative of a low cost and limited computing power device for industrial applications. It is an industrial-grade device: It has no moving parts, is mountable on a DIN rail and has a 24 V power connection. Furthermore, the controller can be equipped with various I/O cards for industrial sensors and actuators including 4-20 mA current signals.

The second system used for the evaluation is a Raspberry Pi Model B with an ARM1176 CPU running at 700 MHz and 512 MB of RAM. Raspberry Pi is a low-cost single-board computer that is widely spread in the education community and often serves as a home automation unit. The tested system runs Raspbian Linux with a 3.12.35 kernel.

The third system, evaluated for comparability purposes, is a typical consumer personal computer with an Intel Core i5-2520M CPU running at 2.5 GHz with 8 GB of RAM. The system runs Ubuntu Linux with a 3.13.0 kernel.

The tested systems ran open62541-based OPC UA servers which responded to the client requests either by using a standard compliant protocol or by using the stateless extension. Only stateless requests were sent over UDP protocol, i.e., the resource-efficient option from Table II was used. Clients performing requests and time measurements used the same stack and ran on a dedicated desktop PC. No security profile and no chunking were used for the measurements. The devices were interconnected via switched Fast Ethernet. The average ping delay between the client and each particular system was 1.1 ms, 0.9 ms and 0.8 ms, respectively. These values allow the estimation of platform differences with respect to used CPU architectures, NICs and buses.

C. Measurement Results and Discussion

The first measured performance indicator is the duration of a complete message exchange, i.e., establishing a connection, sending a single *Read* request querying for a single node property, receiving the service response and closing the connection. All in all, we measure the duration of requests that are depicted in Fig. 4. The duration of a standard compliant message exchange in Fig. 4a is compared to the proposed protocol modifications as depicted in Fig. 4b. The times of the performed measurements can be found in Table III.

We observe that there is a speedup of about 3-4 times when comparing stateful and stateless requests over TCP and a speedup of about 5-7 times when comparing stateful TCP and stateless UDP requests. Note that the presented results were obtained while being subjected to an almost perfect network connectivity with a maximum RTT of 1 ms. Due to the large number of exchanged packets for requests over TCP, the speedup is expected to rise with increasing network latency.

TABLE III: Average duration of connection establishment and single invocation of *Read* service in ms (10000 samples each). "S" and "L" stand for standard compliant or stateless AL, respectively. "T" and "U" stand for TCP or UDP, respectively.

Device	Mode	Mean + 95% CI	Min.	Max.	σ
x86 PC	S T	5.33 [5.32, 5.33]	4.05	7.45	0.22
	L T	1.34 [1.34, 1.34]	0.80	2.60	0.05
	L U	0.77 [0.77, 0.77]	0.58	2.10	0.06
Raspberry Pi	S T	8.33 [8.31, 8.37]	7.21	27.90	1.42
	L T	2.63 [2.62, 2.65]	2.15	12.16	0.64
	L U	1.12 [1.12, 1.13]	0.83	11.60	0.36
WAGO-750	S T	55.36 [55.33, 55.40]	46.77	76.79	1.83
	L T	18.45 [18.41, 18.48]	9.03	24.46	1.56
	L U	10.66 [10.64, 10.67]	6.87	15.89	1.16

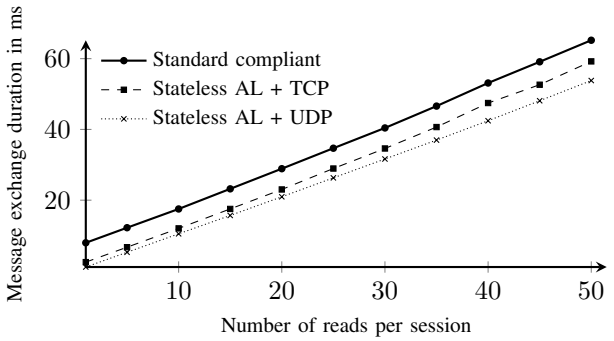


Fig. 5: Total message exchange duration depending on the number of *Read* requests in one exchange on a Raspberry Pi (average of 10000 samples each).

In the second scenario, we measured the duration of the message exchanges including a variable number of *Read* service invocations on the Raspberry Pi. Again, the standard compliant message exchange was compared to the proposed stateless exchanges via TCP and UDP. The results are shown in Fig. 5.

The results are not surprising: An almost linear scaling of the exchange duration depending on the number of requests can be observed. Stateless exchange over UDP takes just a minimal time to perform. Stateless requests over TCP are almost as fast as over UDP (about 4 ms difference in time). Standard compliant communication over TCP leads to the biggest delay while executing a service request (around 10 ms slower than stateless requests over UDP). We also observe that the offsets are nearly constant. The relative speedup decreases as the number of requests per session rises. The speedup of RESTful extensions is, therefore, mostly significant for scenarios with short-lived communication (as stated in Section IV-B1).

In the third scenario, the scalability of the server running on the Raspberry Pi was evaluated for the proposed extensions

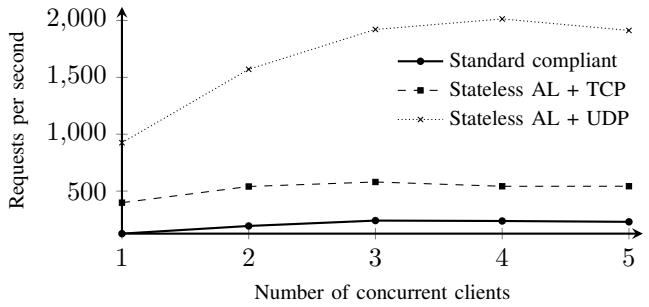


Fig. 6: Server throughput depending on the number of concurrent clients performing *Read* requests (Raspberry Pi). In the standard compliant case, a new session is created for every *Read* to simulate short-lived interactions.

and compared with the standard protocol. A single-threaded server was queried by clients running in a variable number of threads on the client PC. Every client thread established a session, performed one node's attribute read and closed the session in a loop. Clients were executed in parallel for an overall time period of 30 seconds, of which the number of successful iterations was measured in the interval from seconds 8 to 22. The measured throughput is depicted in Fig. 6.

In this scenario the relative throughput compared to the number of threads is the major focus. In the first case, which involved the standard protocol over TCP, the performance peak of about 240 requests per second was achieved with three threads. For stateless TCP connections, the maximal throughput of about 580 requests per second was also reached with three client threads. Finally, with stateless UDP protocol modifications, the top performance of about 2000 requests per second was reached with four threads. In summary, we observe a similar performance improvement as in the second row of Table III – stateless connections over TCP and UDP are roughly 2 and 8 times faster than stateful connections over TCP, respectively.

VII. Summary and Outlook

In this paper we firstly presented an introduction to the RESTful architecture style in industrial applications and then described one application by proposing a RESTful extension for OPC UA. The extension is comprised of two modifications of the current standard that are required by REST: stateless service invocation and caching headers in the service responses. In addition, UDP transport layer for OPC UA has been evaluated with a focus on the reduction of communication overheads.

The benefits and drawbacks of our approach can be summed up as follows: In comparison to standard OPC UA protocol, the stateless extension has several benefits. On the one hand, the communication effort is reduced in terms of exchanged packets between client and server. On the other hand, it scales well on servers with a high amount of connected devices as the server does not need to store any session information. Furthermore, it is possible to build up an infrastructure to cache idempotent requests with the RESTful approach. However, there are also some drawbacks: With the current approach we do not address security aspects at all, even though it is possible to add security on the transport layer. Stateless requests must be transferred in a single OPC UA message. That averts the OPC UA chunking mechanism for messages which are bigger than a maximum allowed message size. This disadvantage is partially remedied by negotiating a large maximum message size, e.g., 65 kB. Despite these disadvantages, we have outlined scenarios where RESTful extensions are preferable over the standard protocol.

The proposed extensions were implemented in an open source OPC UA communication stack called open62541 and evaluated on two embedded systems including an industrial fieldbus controller. The evaluation shows that stateless UDP requests result in an improvement by a factor of up to eight in terms of service invocation time and throughput enhancement.

The presented extensions are not only fully reverse compatible with the standard, but can also be implemented in a couple of dozen lines of code into existing OPC UA servers and clients. This simplifies a transition to RESTful architecture styles for existing OPC UA applications and allows mixed-mode operation.

Working groups of the OPC Foundation are currently working on the implementation of one-to-many and many-to-many publish/subscribe interaction mechanisms and real-time capabilities for OPC UA. The considered solution for publisher/subscribe interaction is to transmit OPC UA data types as the payload of message queue mechanism, such as the Advanced Message Queuing Protocol (AMQP) or a custom UDP-based solution. Real-time capabilities shall be achieved by integrating the upcoming IEEE 802.1 Time-Sensitive Networking standard into OPC UA.

Regarding REST in industrial applications, the emergence of standardized resource representations remains an important topic. While the existing companion standards and the introspection capabilities of OPC UA allow the discovery of information models and their semantics, they do not reduce the need for manual engineering in practice so far. That is, users still integrate system components with out-of-band interface information. A remedy against this are standardized formats for resource self-description, i.e., a generic model for structural and behavioral properties that can be freely parameterized and combined, representing the "greatest common divisor" of applications in a specific domain. Ongoing work in this direction focuses, for example, on the integration of AutomationML and OPC UA in an upcoming companion specification [24], linking structural descriptions, 3D-Models and PLC-controlled dynamic behavior and the description of resource via their skills or generic technical functionality, hiding the details of the underlying implementation [25].

Regarding the technical details of the proposed OPC UA extensions, there is ongoing work on: utilizing advanced caching concepts as defined for HTTP, implementing and evaluating the RESTful extension set on another (open source) OPC UA stack, evaluating the advantages of RESTful architectures (in particular the scalability that is achievable by caching/load balancing) on a real-life application architecture and possible security concepts for RESTful OPC UA. Furthermore, the proposed extensions should be incorporated into the standardization of OPC UA.

REFERENCES

[1] L. Atzori, A. Iera, and G. Morabito, "The internet of things: A survey," *Computer networks*, vol. 54, no. 15, pp. 2787–2805, 2010.

[2] P. Gaj, J. Jasperneite, and M. Felser, "Computer communication within industrial distributed environment – a survey," *Industrial Informatics, IEEE Transactions on*, vol. 9, no. 1, pp. 182–189, 2013.

[3] S. Grüner, J. Pfrommer, and F. Palm, "A RESTful extension of OPC UA," in *Factory Communication Systems (WFCS), 11th IEEE World Conference on*, May 2015, pp. 25–27.

[4] F. Jammes and H. Smit, "Service-oriented paradigms in industrial automation," *Industrial Informatics, IEEE Transactions on*, vol. 1, no. 1, pp. 62–70, Feb 2005.

[5] L. M. S. De Souza, P. Spiess, D. Guinard, M. Köhler, S. Karnouskos, and D. Savio, "Socrades: A web service based shop floor integration infrastructure," in *The internet of things*. Springer, 2008, pp. 50–67.

[6] N. Komoda, "Service oriented architecture (SOA) in industrial systems," in *Industrial Informatics, IEEE International Conference on*, Aug 2006, pp. 1–5.

[7] G. Veiga, J. Pires, and K. Nilsson, "Experiments with service-oriented architectures for industrial robotic cells programming," *Robotics and Computer-Integrated Manufacturing*, vol. 25, no. 4, pp. 746–755, 2009.

[8] G. Cândido, F. Jammes, J. B. de Oliveira, and A. W. Colombo, "SOA at device level in the industrial domain: Assessment of OPC UA and DPWS specifications," in *Industrial Informatics (INDIN), 8th IEEE International Conference on*. IEEE, 2010, pp. 598–603.

[9] S. Cavalieri and G. Cutuli, "Performance evaluation of OPC UA," in *Emerging Technologies and Factory Automation (ETFA), IEEE Conference on*, Sept 2010, pp. 1–8.

[10] T. Luckenbach, P. Gober, S. Arbanowski, A. Kotsopoulos, and K. Kim, "TinyREST-a protocol for integrating sensor networks into the internet," in *Proc. of REALWSN*, 2005, pp. 101–105.

[11] D. Guinard, V. Trifa, and E. Wilde, "A resource oriented architecture for the web of things," in *Internet of Things (IOT), 2010*. IEEE, 2010, pp. 1–8.

[12] C. Bormann, A. P. Castellani, and Z. Shelby, "CoAP: An application protocol for billions of tiny internet nodes," *IEEE Internet Computing*, vol. 16, no. 2, pp. 62–67, 2012.

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

[14] L. Richardson and S. Ruby, *RESTful web services*. O'Reilly, 2007.

[15] M. Collina, G. Corazza, and A. Vanelli-Coralli, "Introducing the QEST broker: Scaling the IoT by bridging MQTT and REST," in *Personal Indoor and Mobile Radio Communications (PIMRC), IEEE 23rd International Symposium on*, Sept 2012, pp. 36–41.

[16] Projexsys, Inc., *HyperUA*. [Online]. Available: http://http://projexsys.com/hyperua/

[17] G. M. Shrestha, J. Imtiaz, and J. Jasperneite, "An optimized OPC UA transport profile to bringing bluetooth low energy device into IP networks," in *Emerging Technologies & Factory Automation (ETFA), IEEE 18th Conference on*. IEEE, 2013, pp. 1–5.

[18] W. Mahnke, S.-H. Leitner, and M. Damm, *OPC unified architecture*, 1st ed. Springer Publishing Company, Incorporated, 2009.

[19] S.-H. Leitner and W. Mahnke, "OPC UA–service-oriented architecture for industrial applications," *ABB Corporate Research Center*, 2006.

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

[21] R. Roman, C. Alcaraz, J. Lopez, and N. Sklavos, "Key management systems for sensor networks in the context of the internet of things," *Computers & Electrical Engineering*, vol. 37, no. 2, pp. 147 – 159, 2011, modern Trends in Applied Security: Architectures, Implementations and Applications. [Online]. Available: http://www.sciencedirect.com/science/article/pii/S0045790611000176

[22] A. Sehgal, V. Perelman, S. Kuryla, and J. Schonwalder, "Management of resource constrained devices in the internet of things," *Communications Magazine, IEEE*, vol. 50, no. 12, pp. 144–149, December 2012.

[23] open62541 project. [Online]. Available: http://www.open62541.org

[24] R. Henßen and M. Schleipen, "Interoperability between OPC UA and AutomationML," *Procedia CIRP*, vol. 25, pp. 297–304, 2014.

[25] J. Pfrommer, D. Stogl, K. Aleksandrov, S. E. Navarro, B. Hein, and J. Beyerer, "Plug & produce by modelling skills and service-oriented orchestration of reconfigurable manufacturing systems," *at Automatisierungstechnik*, vol. 10, no. 63, 2015.

Sten Grüner obtained a Dipl.-Inform. degree in computer science and a Dipl.-Wirt.Inform. degree in business informatics from RWTH Aachen University, Aachen, Germany, in 2011 and 2012, respectively. He is currently employed as a research assistant at the Chair of Process Control Engineering at RWTH Aachen University. His research interests include modeling and implementation of industrial runtime environments, rule-based engineering systems and industrial communication.

Julius Pfrommer obtained a diploma in industrial engineering from Grenoble Institute of Technology, Grenoble, France, in 2012 and a Dipl.-Wirt.Ing. degree from Karlsruhe Institute of Technology (KIT), Karlsruhe, Germany, in 2013. He is currently employed as a research assistant at the Vision and Fusion Laboratory at KIT and works in close collaboration with Fraunhofer IOSB. His research interests include communication in smart automation systems and decentralized control in discrete and continuous settings.

Florian Palm obtained a Master of Science degree in automation engineering from RWTH Aachen University, Aachen, Germany, in 2013. He is currently employed as a research assistant at the Chair of Process Control Engineering at RWTH Aachen University. His research interests include industrial communications, procedure description languages and control engineering.