Network Working Group A. Keranen
Internet-Draft Ericsson
Intended status: Informational M. Kovatsch
Expires: 12 November 2020 Siemens AG
K. Hartke
Ericsson
11 May 2020

RESTful Design for Internet of Things Systems
draft-irtf-t2trg-rest-iot-06

Abstract

This document gives guidance for designing Internet of Things (IoT) systems that follow the principles of the Representational State Transfer (REST) architectural style. This document is a product of the IRTF Thing-to-Thing Research Group (T2TRG).

Status of This Memo

This Internet-Draft will expire on 12 November 2020.

Copyright Notice

Table of Contents

1. Introduction

The Representational State Transfer (REST) architectural style [REST] is a set of guidelines and best practices for building distributed hypermedia systems. At its core is a set of constraints, which when fulfilled enable desirable properties for distributed software systems such as scalability and modifiability. When REST principles are applied to the design of a system, the result is often called RESTful and in particular an API following these principles is called a RESTful API.

Different protocols can be used with RESTful systems, but at the time of writing the most common protocols are HTTP [RFC7230] and CoAP [RFC7252]. Since RESTful APIs are often simple and lightweight, they are a good fit for various IoT applications. The goal of this document is to give basic guidance for designing RESTful systems and APIs for IoT applications and give pointers for more information.

Design of a good RESTful IoT system has naturally many commonalities with other Web systems. Compared to other systems, the key characteristics of many RESTful IoT systems include:

* need to accommodate for constrained devices, so with IoT, REST is not only used for scaling out (large number of clients on a web server), but also for scaling down (efficient server on constrained node)

* data formats, interaction patterns, and other mechanisms that minimize, or preferably avoid, the need for human interaction

* for some classes [RFC7228] of server endpoints, significant constraints, e.g., in energy consumption, and thus implementation complexity, may apply

* endpoints are commonly both clients and servers

* preference for compact and simple data formats to facilitate efficient transfer over (often) constrained networks and lightweight processing in constrained nodes

* the usually large number of endpoints can not be updated simultaneously, yet the system needs to be able to evolve in the field without long downtimes

2. Terminology

This section explains some of the common terminology that is used in the context of RESTful design for IoT systems. For terminology of constrained nodes and networks, see [RFC7228].

Cache: A local store of response messages and the subsystem that controls storage, retrieval, and deletion of messages in it.

Client: A node that sends requests to servers and receives responses. In RESTful IoT systems it's common for nodes to have more than one role (e.g., both server and client; see Section 3.1).

Client State: The state kept by a client between requests. This typically includes the currently processed representation, the set of active requests, the history of requests, bookmarks (URIs stored for later retrieval), and application-specific state (e.g., local variables). (Note that this is called "Application State" in [REST], which has some ambiguity in modern (IoT) systems where the overall state of the distributed application (i.e., application state) is reflected in the union of all Client States and Resource States of all clients and servers involved.)

Content Negotiation: The practice of determining the "best" representation for a client when examining the current state of a resource. The most common forms of content negotiation are Proactive Content Negotiation and Reactive Content Negotiation.

Dereference: To use an access mechanism (e.g., HTTP or CoAP) to perform an action on a URI's resource.

Dereferencable URI: A URI that can be dereferenced, i.e., an action can be performed on the identified resource. Not all HTTP or CoAP URIs are dereferencable, e.g., when the target resource does not exist.

Form: A hypermedia control that enables a client to change the state of a resource or to construct a query locally.

Forward Proxy: An intermediary that is selected by a client, usually via local configuration rules, and that can be tasked to make requests on behalf of the client. This may be useful, for example, when the client lacks the capability to make the request itself or to service the response from a cache in order to reduce response time, network bandwidth, and energy consumption.

Gateway: A reverse proxy that provides an interface to a non-RESTful system such as legacy systems or alternative technologies such as Bluetooth Attribute Profile (ATT) or Generic Attribute Profile (GATT). See also "Reverse Proxy".

Hypermedia Control: A component, such as a link or a form, embedded in a representation that identifies a resource for future hypermedia interactions. If the client engages in an interaction with the identified resource, the result may be a change to resource state and/or client state.

Idempotent Method: A method where multiple identical requests with that method lead to the same visible resource state as a single such request.

Link: A hypermedia control that enables a client to navigate between resources and thereby change the client state.

Link Relation Type: An identifier that describes how the link target resource relates to the current resource (see [RFC5988]).

Media Type: A string such as "text/html" or "application/json" that is used to label representations so that it is known how the representation should be interpreted and how it is encoded.

Method: An operation associated with a resource. Common methods include GET, PUT, POST, and DELETE (see Section 3.5 for details).

Origin Server: A server that is the definitive source for representations of its resources and the ultimate recipient of any request that intends to modify its resources. In contrast, intermediaries (such as proxies caching a representation) can assume the role of a server, but are not the source for representations as these are acquired from the origin server.

Proactive Content Negotiation: A content negotiation mechanism where the server selects a representation based on the expressed preference of the client. For example, an IoT application could send a request to a sensor with preferred media type "application/senml+json".

Reactive Content Negotiation: A content negotiation mechanism where the client selects a representation from a list of available representations. The list may, for example, be included by a server in an initial response. If the user agent is not satisfied by the initial response representation, it can request one or more of the alternative representations, selected based on metadata (e.g., available media types) included in the response.

Representation: A serialization that represents the current or intended state of a resource and that can be transferred between clients and servers. REST requires representations to be self-describing, meaning that there must be metadata that allows peers to understand which representation format is used. Depending on the protocol needs and capabilities, there can be additional metadata that is transmitted along with the representation.

Representation Format: A set of rules for serializing resource state. On the Web, the most prevalent representation format is HTML. Other common formats include plain text and formats based on JSON [RFC8259], XML, or RDF. Within IoT systems, often compact formats based on JSON, CBOR [RFC7049], and EXI [W3C.REC-exi-20110310] are used.

Representational State Transfer (REST): An architectural style for Internet-scale distributed hypermedia systems.

Resource: An item of interest identified by a URI. Anything that can be named can be a resource. A resource often encapsulates a piece of state in a system. Typical resources in an IoT system can be, e.g., a sensor, the current value of a sensor, the location of a device, or the current state of an actuator.

Resource State: A model of a resource's possible states that is represented in a supported representation type, typically a media type. Resources can change state because of REST interactions with them, or they can change state for reasons outside of the REST model.

Resource Type: An identifier that annotates the application-semantics of a resource (see Section 3.1 of [RFC6690]).

Reverse Proxy: An intermediary that appears as a server towards the client but satisfies the requests by forwarding them to the actual server (possibly via one or more other intermediaries). A reverse proxy is often used to encapsulate legacy services, to improve server performance through caching, and to enable load balancing across multiple machines.

Safe Method: A method that does not result in any state change on the origin server when applied to a resource.

Server: A node that listens for requests, performs the requested operation and sends responses back to the clients.

Transfer protocols: In particular in IoT domain, protocols above transport layer that are used to transfer data objects and provide semantics for operations on the data.

Transfer layer: Re-usable part of the application layer used to transfer the application specific data items using a standard set of methods that can fulfill application-specific operations.

Uniform Resource Identifier (URI): A global identifier for resources. See Section 3.3 for more details.

3. Basics

3.1. Architecture

The components of a RESTful system are assigned one or both of two roles: client or server. Note that the terms "client" and "server" refer only to the roles that the nodes assume for a particular message exchange. The same node might act as a client in some communications and a server in others. Classic user agents (e.g., Web browsers) are always in the client role and have the initiative to issue requests. Origin servers always have the server role and govern over the resources they host. Simple IoT devices, such as sensors and actuators, are commonly acting as servers and exposing their physical world interaction capabilities (e.g., temperature measurement or door lock control capability) as resources. Typical IoT system client can be a cloud service that retrieves data from the sensors and commands the actuators based on the sensor information. Alternatively an IoT data storage system could work as a server where IoT sensor devices send data, in client role.

```
                ________                       _________
               |        |                     |         |
               | User  (C)-------------------(S) Origin |
               | Agent  |                     |  Server |
               |________|                     |_________|
               (Browser)                      (Web Server)
```

Figure 1: Client-Server Communication

Intermediaries (such as forward proxies, reverse proxies, and gateways) implement both roles, but only forward requests to other intermediaries or origin servers. They can also translate requests to different protocols, for instance, as CoAP-HTTP cross-proxies [RFC8075].

```
     ________       __________                        _________
    |        |     |          |                      |         |
    | User  (C)---(S) Inter- (C)--------------------(S) Origin |
    | Agent  |     |  mediary |                      |  Server |
    |________|     |__________|                      |_________|
    (Browser)     (Forward Proxy)                    (Web Server)
```

Figure 2: Communication with Forward Proxy

Reverse proxies are usually imposed by the origin server. In addition to the features of a forward proxy, they can also provide an interface for non-RESTful services such as legacy systems or alternative technologies such as Bluetooth ATT/GATT. In this case, reverse proxies are usually called gateways. This property is enabled by the Layered System constraint of REST, which says that a client cannot see beyond the server it is connected to (i.e., it is left unaware of the protocol/paradigm change).

```
    ________                        __________       _________
   |        |                      |          |     |         |
   | User  (C)--------------------(S) Inter- (x)---(x) Origin |
   | Agent  |                      |  mediary |     |  Server |
   |________|                      |__________|     |_________|
   (Browser)                        (Gateway)     (Legacy System)
```

Figure 3: Communication with Reverse Proxy

Nodes in IoT systems often implement both roles. Unlike intermediaries, however, they can take the initiative as a client (e.g., to register with a directory, such as CoRE Resource Directory [I-D.ietf-core-resource-directory], or to interact with another thing) and act as origin server at the same time (e.g., to serve sensor values or provide an actuator interface).

```
      ________                                         _________
     |        |                                       |         |
     | Thing (C)-------------------------------------(S) Origin |
     |       (S)                                      |  Server |
     |________| \                                     |_________|
      (Sensor)   \   ________                     (Resource Directory)
                  \ |        |
                   (C) Thing |
                    |________|
                   (Controller)
```

Figure 4: Constrained RESTful environments

3.2. System design

When designing a RESTful system, the primary effort goes into modeling the state of the distributed application and assigning it to the different components (i.e., clients and servers). How clients can navigate through the resources and modify state to achieve their goals is defined through hypermedia controls, that is, links and forms. Hypermedia controls span a kind of a state machine where the nodes are resources and the transitions are links or forms. Clients run this state machine (i.e., the application) by retrieving representations, processing the data, and following the included hypermedia controls. In REST, remote state is changed by submitting forms. This is usually done by retrieving the current state, modifying the state on the client side, and transferring the new state to the server in the form of new representations - rather than calling a service and modifying the state on the server side.

Client state encompasses the current state of the described state machine and the possible next transitions derived from the hypermedia controls within the currently processed representation (see Section 2). Furthermore, clients can have part of the state of the distributed application in local variables.

Resource state includes the more persistent data of an application (i.e., independent of individual clients). This can be static data such as device descriptions, persistent data such as system configurations, but also dynamic data such as the current value of a sensor on a thing.

It is important to distinguish between "client state" and "resource state" and keep them separate. Following the Stateless constraint, the client state must be kept only on clients. That is, there is no establishment of shared information about past and future interactions between client and server (usually called a session). On the one hand, this makes requests a bit more verbose since every

request must contain all the information necessary to process it. On the other hand, this makes servers efficient and scalable, since they do not have to keep any state about their clients. Requests can easily be distributed over multiple worker threads or server instances. For IoT systems, this constraint lowers the memory requirements for server implementations, which is particularly important for constrained servers (e.g., sensor nodes) and servers serving large amount of clients (e.g., Resource Directory).

3.3. Uniform Resource Identifiers (URIs)

An important part of RESTful API design is to model the system as a set of resources whose state can be retrieved and/or modified and where resources can be potentially also created and/or deleted.

Uniform Resource Identifiers (URIs) are used to indicate a resource for interaction, to reference a resource from another resource, to advertise or bookmark a resource, or to index a resource by search engines.

```
  foo://example.com:8042/over/there?name=ferret#nose
  \_/   \______________/\_________/ \_________/ \__/
   |           |            |            |        |
scheme     authority       path        query   fragment
```

A URI is a sequence of characters that matches the syntax defined in [RFC3986]. It consists of a hierarchical sequence of five components: scheme, authority, path, query, and fragment (from most significant to least significant). A scheme creates a namespace for resources and defines how the following components identify a resource within that namespace. The authority identifies an entity that governs part of the namespace, such as the server "www.example.org" in the "http" scheme. A host name (e.g., a fully qualified domain name) or an IP address, potentially followed by a transport layer port number, are usually used in the authority component for the "http" and "coap" schemes. The path and query contain data to identify a resource within the scope of the URI's scheme and naming authority. The fragment allows to refer to some portion of the resource, such as a Record in a SenML Pack (Section 9 of [RFC8428]). However, fragments are processed only at client side and not sent on the wire. [RFC7320] provides more details on URI design and ownership with best current practices for establishing URI structures, conventions, and formats.

For RESTful IoT applications, typical schemes include "https", "coaps", "http", and "coap". These refer to HTTP and CoAP, with and without Transport Layer Security (TLS, [RFC5246] for TLS 1.2 and [RFC8446] for TLS 1.3). (CoAP uses Datagram TLS (DTLS) [RFC6347],

the variant of TLS for UDP.) These four schemes also provide means for locating the resource; using the HTTP protocol for "http" and "https", and with the CoAP protocol for "coap" and "coaps". If the scheme is different for two URIs (e.g., "coap" vs. "coaps"), it is important to note that even if the rest of the URI is identical, these are two different resources, in two distinct namespaces.

Some schemes are for URIs with main purpose as identifiers and hence are not dereferencable, e.g., the "urn" scheme can be used to construct unique names in registered namespaces. In particular the "urn:dev" [I-D.ietf-core-dev-urn] details multiple ways for generating and representing endpoint identifiers of IoT devices.

The query parameters can be used to parametrize the resource. For example, a GET request may use query parameters to request the server to send only certain kind data of the resource (i.e., filtering the response). Query parameters in PUT and POST requests do not have such established semantics and are not commonly used. Whether the order of the query parameters matters in URIs is unspecified and they can be re-ordered e.g., by proxies. Therefore applications should not rely on their order; see Section 3.3 of [RFC6943] for more details.

Due to the relatively complex processing rules and text representation format, URI handling can be difficult to implement correctly in constrained devices. Constrained Resource Identifiers [I-D.ietf-core-href] provide a CBOR-based format of URIs that is better suited also for resource constrained IoT devices.

3.4. Representations

Clients can retrieve the resource state from an origin server or manipulate resource state on the origin server by transferring resource representations. Resource representations have a content-type (media-type, optionally with parameters) that tells how the representation should be interpreted by identifying the representation format used.

Typical media-types for IoT systems include:

* "text/plain" for simple UTF-8 text

* "application/octet-stream" for arbitrary binary data

* "application/json" for the JSON format [RFC8259]

* "application/cbor" for CBOR [RFC7049]

- "application/exi" for EXI [W3C.REC-exi-20110310]

- "application/link-format" for CoRE Link Format [RFC6690]

- "application/senml+json" and "application/senml+cbor" for Sensor Measurement Lists (SenML) data [RFC8428]

A full list of registered Internet Media Types is available at the IANA registry [IANA-media-types] and numerical identifiers for media-types, parameters, and content-codings registered for use with CoAP are listed at CoAP Content-Formats IANA registry [IANA-CoAP-media].

The terms "media-type", "content-type", and "content-format" (short identifier of content-type and content-coding, abbreviated for historical reasons "ct") are often used when referring to representation formats used with CoAP. The differences between these terms are discussed in more detail in [I-D.bormann-core-media-content-type-format].

3.5. HTTP/CoAP Methods

Section 4.3 of [RFC7231] defines the set of methods in HTTP; Section 5.8 of [RFC7252] defines the set of methods in CoAP. As part of the Uniform Interface constraint, each method can have certain properties that give guarantees to clients.

Safe methods do not cause any state change on the origin server when applied to a resource. For example, the GET method only returns a representation of the resource state but does not change the resource. Thus, it is always safe for a client to retrieve a representation without affecting server-side state.

Idempotent methods can be applied multiple times to the same resource while causing the same visible resource state as a single such request. For example, the PUT method replaces the state of a resource with a new state; replacing the state multiple times with the same new state still results in the same state for the resource. However, the response from the server can be different when the same idempotent method is used multiple times. For example when DELETE is used twice on an existing resource, the first request would remove the association and return success acknowledgement whereas the second request would likely result in error response due to non-existing resource.

The following lists the most relevant methods and gives a short explanation of their semantics.

3.5.1. GET

The GET method requests a current representation for the target resource, while the origin server must ensure that there are no side-effects on the resource state. Only the origin server needs to know how each of its resource identifiers corresponds to an implementation and how each implementation manages to select and send a current representation of the target resource in a response to GET.

A payload within a GET request message has no defined semantics.

The GET method is safe and idempotent.

3.5.2. POST

The POST method requests that the target resource process the representation enclosed in the request according to the resource's own specific semantics.

If one or more resources has been created on the origin server as a result of successfully processing a POST request, the origin server sends a 201 (Created) response containing a Location header field (with HTTP) or Location-Path and/or Location-Query Options (with CoAP) that provide an identifier for the resource created. The server also includes a representation that describes the status of the request while referring to the new resource(s).

The POST method is not safe nor idempotent.

3.5.3. PUT

The PUT method requests that the state of the target resource be created or replaced with the state defined by the representation enclosed in the request message payload. A successful PUT of a given representation would suggest that a subsequent GET on that same target resource will result in an equivalent representation being sent.

The fundamental difference between the POST and PUT methods is highlighted by the different intent for the enclosed representation. The target resource in a POST request is intended to handle the enclosed representation according to the resource's own semantics, whereas the enclosed representation in a PUT request is defined as replacing the state of the target resource. Hence, the intent of PUT is idempotent and visible to intermediaries, even though the exact effect is only known by the origin server.

The PUT method is not safe, but is idempotent.

3.5.4. DELETE

The DELETE method requests that the origin server remove the association between the target resource and its current functionality.

If the target resource has one or more current representations, they might or might not be destroyed by the origin server, and the associated storage might or might not be reclaimed, depending entirely on the nature of the resource and its implementation by the origin server.

The DELETE method is not safe, but is idempotent.

3.5.5. FETCH

The CoAP-specific FETCH method [RFC8132] requests a representation of a resource parameterized by a representation enclosed in the request.

The fundamental difference between the GET and FETCH methods is that the request parameters are included as the payload of a FETCH request, while in a GET request they're typically part of the query string of the request URI.

The FETCH method is safe and idempotent.

3.5.6. PATCH

The PATCH method [RFC5789] [RFC8132] requests that a set of changes described in the request entity be applied to the target resource.

The PATCH method is not safe nor idempotent.

The CoAP-specific iPATCH method is a variant of the PATCH method that is not safe, but is idempotent.

3.6. HTTP/CoAP Status/Response Codes

Section 6 of [RFC7231] defines a set of Status Codes in HTTP that are used by application to indicate whether a request was understood and satisfied, and how to interpret the answer. Similarly, Section 5.9 of [RFC7252] defines the set of Response Codes in CoAP.

The status codes consist of three digits (e.g., "404" with HTTP or "4.04" with CoAP) where the first digit expresses the class of the code. Implementations do not need to understand all status codes, but the class of the code must be understood. Codes starting with 1 are informational; the request was received and being processed.

Codes starting with 2 indicate a successful request. Codes starting with 3 indicate redirection; further action is needed to complete the request. Codes stating with 4 and 5 indicate errors. The codes starting with 4 mean client error (e.g., bad syntax in the request) whereas codes starting with 5 mean server error; there was no apparent problem with the request, but server was not able to fulfill the request.

Responses may be stored in a cache to satisfy future, equivalent requests. HTTP and CoAP use two different patterns to decide what responses are cacheable. In HTTP, the cacheability of a response depends on the request method (e.g., responses returned in reply to a GET request are cacheable). In CoAP, the cacheability of a response depends on the response code (e.g., responses with code 2.04 are cacheable). This difference also leads to slightly different semantics for the codes starting with 2; for example, CoAP does not have a 2.00 response code whereas 200 ("OK") is commonly used with HTTP.

4. REST Constraints

The REST architectural style defines a set of constraints for the system design. When all constraints are applied correctly, REST enables architectural properties of key interest [REST]:

* Performance

* Scalability

* Reliability

* Simplicity

* Modifiability

* Visibility

* Portability

The following sub-sections briefly summarize the REST constraints and explain how they enable the listed properties.

4.1. Client-Server

As explained in the Architecture section, RESTful system components have clear roles in every interaction. Clients have the initiative to issue requests, intermediaries can only forward requests, and servers respond requests, while origin servers are the ultimate recipient of requests that intent to modify resource state.

This improves simplicity and visibility (also for digital forensics), as it is clear which component started an interaction. Furthermore, it improves modifiability through a clear separation of concerns.

In IoT systems, endpoints often assume both roles of client and (origin) server simultaneously. When an IoT device has initiative (because there is a user, e.g., pressing a button, or installed rules/policies), it acts as a client. When a device offers a service, it is in server role.

4.2. Stateless

The Stateless constraint requires messages to be self-contained. They must contain all the information to process it, independent from previous messages. This allows to strictly separate the client state from the resource state.

This improves scalability and reliability, since servers or worker threads can be replicated. It also improves visibility because message traces contain all the information to understand the logged interactions. Furthermore, the Stateless constraint enables caching.

For IoT, the scaling properties of REST become particularly important. Note that being self-contained does not necessarily mean that all information has to be inlined. Constrained IoT devices may choose to externalize metadata and hypermedia controls using Web linking, so that only the dynamic content needs to be sent and the static content such as schemas or controls can be cached.

4.3. Cache

This constraint requires responses to have implicit or explicit cache-control metadata. This enables clients and intermediary to store responses and re-use them to locally answer future requests. The cache-control metadata is necessary to decide whether the information in the cached response is still fresh or stale and needs to be discarded.

Cache improves performance, as less data needs to be transferred and response times can be reduced significantly. Less transfers also improves scalability, as origin servers can be protected from too many requests. Local caches furthermore improve reliability, since requests can be answered even if the origin server is temporarily not available.

Caching usually only makes sense when the data is used by multiple participants. In IoT systems, however, it might make sense to cache also individual data to protect constrained devices and networks from frequent requests of data that does not change often. Security often hinders the ability to cache responses. For IoT systems, object security [RFC8613] may be preferable over transport layer security, as it enables intermediaries to cache responses while preserving security.

4.4. Uniform Interface

All RESTful APIs use the same, uniform interface independent of the application. This simple interaction model is enabled by exchanging representations and modifying state locally, which simplifies the interface between clients and servers to a small set of methods to retrieve, update, and delete state - which applies to all applications.

In contrast, in a service-oriented RPC approach, all required ways to modify state need to be modeled explicitly in the interface resulting in a large set of methods - which differs from application to application. Moreover, it is also likely that different parties come up with different ways how to modify state, including the naming of the procedures, while the state within an application is a bit easier to agree on.

A REST interface is fully defined by:

* URIs to identify resources

* representation formats to represent and manipulate resource state

* self-descriptive messages with a standard set of methods (e.g., GET, POST, PUT, DELETE with their guaranteed properties)

* hypermedia controls within representations

The concept of hypermedia controls is also known as HATEOAS: Hypermedia As The Engine Of Application State. The origin server embeds controls for the interface into its representations and thereby informs the client about possible next requests. The most

used control for RESTful systems today is Web Linking [RFC5988]. Hypermedia forms are more powerful controls that describe how to construct more complex requests, including representations to modify resource state.

While this is the most complex constraints (in particular the hypermedia controls), it improves many different key properties. It improves simplicity, as uniform interfaces are easier to understand. The self-descriptive messages improve visibility. The limitation to a known set of representation formats fosters portability. Most of all, however, this constraint is the key to modifiability, as hypermedia-driven, uniform interfaces allow clients and servers to evolve independently, and hence enable a system to evolve.

For a large number of IoT applications, the hypermedia controls are mainly used for the discovery of resources, as they often serve sensor data. Such resources are "dead ends", as they usually do not link any further and only have one form of interaction: fetching the sensor value. For IoT, the critical parts of the Uniform Interface constraint are the descriptions of messages and representation formats used. Simply using, for instance, "application/json" does not help machine clients to understand the semantics of the representation. Yet defining very precise media types limits the re-usability and interoperability. Representation formats such as SenML [RFC8428] try to find a good trade-off between precision and re-usability. Another approach is to combine a generic format such as JSON with syntactic as well as semantic annotations (see [I-D.handrews-json-schema-validation] and [W3C-TD], resp.).

4.5. Layered System

This constraint enforces that a client cannot see beyond the server with which it is interacting.

A layered system is easier to modify, as topology changes become transparent. Furthermore, this helps scalability, as intermediaries such as load balancers can be introduced without changing the client side. The clean separation of concerns helps with simplicity.

IoT systems greatly benefit from this constraint, as it allows to effectively shield constrained devices behind intermediaries and is also the basis for gateways, which are used to integrate other (IoT) ecosystems.

4.6. Code-on-Demand

This principle enables origin servers to ship code to clients.

Code-on-Demand improves modifiability, since new features can be deployed during runtime (e.g., support for a new representation format). It also improves performance, as the server can provide code for local pre-processing before transferring the data.

As of today, code-on-demand has not been explored much in IoT systems. Aspects to consider are that either one or both nodes are constrained and might not have the resources to host or dynamically fetch and execute such code. Moreover, the origin server often has no understanding of the actual application a mashup client realizes. Still, code-on-demand can be useful for small polyfills, e.g., to decode payloads, and potentially other features in the future.

5. Hypermedia-driven Applications

Hypermedia-driven applications take advantage of hypermedia controls, i.e., links and forms, embedded in the resource representations. A hypermedia client is a client that is capable of processing these hypermedia controls. Hypermedia links can be used to give additional information about a resource representation (e.g., the source URI of the representation) or pointing to other resources. The forms can be used to describe the structure of the data that can be sent (e.g., with a POST or PUT method) to a server, or how a data retrieval (e.g., GET) request for a resource should be formed. In a hypermedia-driven application the client interacts with the server using only the hypermedia controls, instead of selecting methods and/or constructing URIs based on out-of-band information, such as API documentation. The Constrained RESTful Application Language (CoRAL) [I-D.ietf-core-coral] provides a hypermedia-format that is suitable for constrained IoT environments.

5.1. Motivation

The advantage of this approach is increased evolvability and extensibility. This is important in scenarios where servers exhibit a range of feature variations, where it's expensive to keep evolving client knowledge and server knowledge in sync all the time, or where there are many different client and server implementations. Hypermedia controls serve as indicators in capability negotiation. In particular, they describe available resources and possible operations on these resources using links and forms, respectively.

There are multiple reasons why a server might introduce new links or forms:

* The server implements a newer version of the application. Older clients ignore the new links and forms, while newer clients are able to take advantage of the new features by following the new links and submitting the new forms.

* The server offers links and forms depending on the current state. The server can tell the client which operations are currently valid and thus help the client navigate the application state machine. The client does not have to have knowledge which operations are allowed in the current state or make a request just to find out that the operation is not valid.

* The server offers links and forms depending on the client's access control rights. If the client is unauthorized to perform a certain operation, then the server can simply omit the links and forms for that operation.

5.2. Knowledge

A client needs to have knowledge of a couple of things for successful interaction with a server. This includes what resources are available, what representations of resource states are available, what each representation describes, how to retrieve a representation, what state changing operations on a resource are possible, how to perform these operations, and so on.

Some part of this knowledge, such as how to retrieve the representation of a resource state, is typically hard-coded in the client software. For other parts, a choice can often be made between hard-coding the knowledge or acquiring it on-demand. The key to success in either case is the use of in-band information for identifying the knowledge that is required. This enables the client to verify that it has all the required knowledge or to acquire missing knowledge on-demand.

A hypermedia-driven application typically uses the following identifiers:

* URI schemes that identify communication protocols,

* Internet Media Types that identify representation formats,

* link relation types or resource types that identify link semantics,

* form relation types that identify form semantics,

* variable names that identify the semantics of variables in templated links, and

* form field names that identify the semantics of form fields in forms.

The knowledge about these identifiers as well as matching implementations have to be shared a priori in a RESTful system.

5.3. Interaction

A client begins interacting with an application through a GET request on an entry point URI. The entry point URI is the only URI a client is expected to know before interacting with an application. From there, the client is expected to make all requests by following links and submitting forms that are provided in previous responses. The entry point URI can be obtained, for example, by manual configuration or some discovery process (e.g., DNS-SD [RFC6763] or Resource Directory [I-D.ietf-core-resource-directory]). For Constrained RESTful environments "/.well-known/core" relative URI is defined as a default entry point for requesting the links hosted by servers with known or discovered addresses [RFC6690].

5.4. Hypermedia-driven Design Guidance

Assuming self-describing representation formats (i.e., human-readable with carefully chosen terms or processible by a formatting tool) and a client supporting the URI scheme used, a good rule of thumb for a good hypermedia-driven design is the following: A developer should only need an entry point URI to drive the application. All further information how to navigate through the application (links) and how to construct more complex requests (forms) are published by the server(s). There must be no need for additional, out-of-band information (e.g., API specification).

For machines, a well-chosen set of information needs to be shared a priori to agree on machine-understandable semantics. Agreeing on the exact semantics of terms for relation types and data elements will of course also help the developer. [I-D.hartke-core-apps] proposes a convention for specifying the set of information in a structured way.

6. Design Patterns

Certain kinds of design problems are often recurring in variety of domains, and often re-usable design patterns can be applied to them. Also some interactions with a RESTful IoT system are straightforward to design; a classic example of reading a temperature from a thermometer device is almost always implemented as a GET request to a resource that represents the current value of the thermometer. However, certain interactions, for example data conversions or event handling, do not have as straightforward and well established ways to represent the logic with resources and REST methods.

The following sections describe how common design problems such as different interactions can be modeled with REST and what are the benefits of different approaches.

6.1. Collections

A common pattern in RESTful systems across different domains is the collection. A collection can be used to combine multiple resources together by providing resources that consist of set of (often partial) representations of resources, called items, and links to resources. The collection resource also defines hypermedia controls for managing and searching the items in the collection.

Examples of the collection pattern in RESTful IoT systems are the CoRE Resource Directory [I-D.ietf-core-resource-directory], CoAP pub/ sub broker [I-D.ietf-core-coap-pubsub], and resource discovery via ".well-known/core". Collection+JSON [CollectionJSON] is an example of a generic collection Media Type.

6.2. Calling a Procedure

To modify resource state, clients usually use GET to retrieve a representation from the server, modify that locally, and transfer the resulting state back to the server with a PUT (see Section 4.4). Sometimes, however, the state can only be modified on the server side, for instance, because representations would be too large to transfer or part of the required information shall not be accessible to clients. In this case, resource state is modified by calling a procedure (or "function"). This is usually modeled with a POST request, as this method leaves the behavior semantics completely to the server. Procedure calls can be divided into two different classes based on how long they are expected to execute: "instantly" returning and long-running.

6.2.1. Instantly Returning Procedures

When the procedure can return within the expected response time of the system, the result can be directly returned in the response. The result can either be actual content or just a confirmation that the call was successful. In either case, the response does not contain a representation of the resource, but a so-called action result. Action results can still have hypermedia controls to provide the possible transitions in the application state machine.

6.2.2. Long-running Procedures

When the procedure takes longer than the expected response time of the system, or even longer than the response timeout, it is a good pattern to create a new resource to track the "task" execution. The server would respond instantly with a "Created" status (HTTP code 201 or CoAP 2.01) and indicate the location of the task resource in the corresponding header field (or CoAP option) or as a link in the action result. The created resource can be used to monitor the progress, to potentially modify queued tasks or cancel tasks, and to eventually retrieve the result.

Monitoring information would be modeled as state of the task resource, and hence be retrievable as representation. The result - when available - can be embedded in the representation or given as a link to another sub-resource. Modifying tasks can be modeled with forms that either update sub-resources via PUT or do a partial write using PATCH or POST. Canceling a task would be modeled with a form that uses DELETE to remove the task resource.

6.2.3. Conversion

A conversion service is a good example where REST resources need to behave more like a procedure call. The knowledge of converting from one representation to another is located only at the server to relieve clients from high processing or storing lots of data. There are different approaches that all depend on the particular conversion problem.

As mentioned in the previous sections, POST request are a good way to model functionality that does not necessarily affect resource state. When the input data for the conversion is small and the conversion result is deterministic, however, it can be better to use a GET request with the input data in the URI query part. The query is parameterizing the conversion resource, so that it acts like a look-up table. The benefit is that results can be cached also for HTTP (where responses to POST are not cacheable). In CoAP, cacheability depends on the response code, so that also a response to a POST request can be made cacheable through a 2.05 Content code.

When the input data is large or has a binary encoding, it is better to use POST requests with a proper Media Type for the input representation. A POST request is also more suitable, when the result is time-dependent and the latest result is expected (e.g., exchange rates).

6.2.4. Events as State

In event-centric paradigms such as pub/sub, events are usually represented by an incoming message that might even be identical for each occurrence. Since the messages are queued, the receiver is aware of each occurrence of the event and can react accordingly. For instance, in an event-centric system, ringing a door bell would result in a message being sent that represents the event that it was rung.

In resource-oriented paradigms such as REST, messages usually carry the current state of the remote resource, independent from the changes (i.e., events) that have lead to that state. In a naive yet natural design, a door bell could be modeled as a resource that can have the states unpressed and pressed. There are, however, a few issues with this approach. Polling (i.e., periodically retrieving) the door bell resource state is not a good option, as the client is highly unlikely to be able to observe all the changes in the pressed state with any realistic polling interval. When using CoAP Observe with Confirmable notifications, the server will usually send two notifications for the event that the door bell was pressed: notification for changing from unpressed to pressed and another one for changing back to unpressed. If the time between the state changes is very short, the server might drop the first notification, as Observe only guarantees only eventual consistency (see Section 1.3 of [RFC7641]).

The solution is to pick a state model that fits better to the application. In the case of the door bell - and many other event-driven resources - the solution could be a counter that counts how often the bell was pressed. The corresponding action is taken each

time the client observes a change in the received representation. In the case of a network outage, this could lead to a ringing sound long after the bell was rung. Also including a timestamp of the last counter increment in the state can help to suppress ringing a sound when the event has become obsolete. Another solution would be to change the client/server roles of the door bell button and the ringer, as described in Section 6.3.

6.3. Server Push

Overall, a universal mechanism for server push, that is, change-of-state notifications and stand-alone event notifications, is still an open issue that is being discussed in the Thing-to-Thing Research Group. It is connected to the state-event duality problem and custody transfer, that is, the transfer of the responsibility that a message (e.g., event) is delivered successfully.

A proficient mechanism for change-of-state notifications is currently only available for CoAP: Observing resources [RFC7641]. The CoAP Observe mechanism offers eventual consistency, which guarantees "that if the resource does not undergo a new change in state, eventually all registered observers will have a current representation of the latest resource state". It intrinsically deals with the challenges of lossy networks, where notifications might be lost, and constrained networks, where there might not be enough bandwidth to propagate all changes.

For stand-alone event notifications, that is, where every single notification contains an identifiable event that must not be lost, observing resources is not a good fit. A better strategy is to model each event as a new resource, whose existence is notified through change-of-state notifications of an index resource (cf. Collection pattern). Large numbers of events will cause the notification to grow large, as it needs to contain a large number of Web links. Block-wise transfers [RFC7959] can help here. When the links are ordered by freshness of the events, the first block can already contain all links to new events. Then, observers do not need to retrieve the remaining blocks from the server, but only the representations of the new event resources.

An alternative pattern is to exploit the dual roles of IoT devices, in particular when using CoAP: they are usually client and server at the same time. An endpoint interested in observing the events would subscribe to them by registering a callback URI at the origin server, e.g., using a POST request with the URI or a hypermedia document in the payload, and receiving the location of a temporary "subscription resource" as handle in the response. The origin server would then publish events by sending requests containing the event data to the

observer's callback URI; here POST can be used to add events to a collection located at the callback URI or PUT can be used when the event data is a new state that shall replace the outdated state at the callback URI. The cancellation can be modeled through deleting the subscription resource. This pattern makes the origin server responsible for delivering the event notifications. This goes beyond retransmissions of messages; the origin server is usually supposed to queue all undelivered events and to retry until successful delivery or explicit cancellation. In HTTP, this pattern is known as REST Hooks.

Methods for configuring server push and notification conditions with CoAP are provided by the CoRE Dynamic Resource Linking specification [I-D.ietf-core-dynlink].

In HTTP, there exist a number of workarounds to enable server push, e.g., long polling and streaming [RFC6202] or server-sent events [W3C.REC-html5-20141028]. In IoT systems, long polling can introduce a considerable overhead, as the request has to be repeated for each notification. Streaming and server-sent events (the latter is actually an evolution of the former) are more efficient, as only one request is sent. However, there is only one response header and subsequent notifications can only have content. Individual status and metadata needs to be included in the content message. This reduces HTTP again to a pure transport, as its status signaling and metadata capabilities cannot be used.

7. Security Considerations

This document does not define new functionality and therefore does not introduce new security concerns. We assume that system designers apply classic Web security on top of the basic RESTful guidance given in this document. Thus, security protocols and considerations from related specifications apply to RESTful IoT design. These include:

* Transport Layer Security (TLS): [RFC8446], [RFC5246], and [RFC6347]

* Internet X.509 Public Key Infrastructure: [RFC5280]

* HTTP security: Section 9 of [RFC7230], Section 9 of [RFC7231], etc.

* CoAP security: Section 11 of [RFC7252]

* URI security: Section 7 of [RFC3986]

IoT-specific security is active area of standardization at the time of writing. First finalized specifications include:

* (D)TLS Profiles for the Internet of Things: [RFC7925]

* CBOR Object Signing and Encryption (COSE) [RFC8152]

* CBOR Web Token [RFC8392]

* Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs) [RFC8747]

* Object Security for Constrained RESTful Environments (OSCORE) [RFC8613]

* Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework [I-D.ietf-ace-oauth-authz]

* ACE profiles for DTLS [I-D.ietf-ace-dtls-authorize] and OSCORE [I-D.ietf-ace-oscore-profile]

Further IoT security considerations are available in [RFC8576].

8. Acknowledgement

The authors would like to thank Mike Amundsen, Heidi-Maria Back, Carsten Bormann, Tero Kauppinen, Michael Koster, Mert Ocak, Robby Simpson, Ravi Subramaniam, Dave Thaler, Niklas Widell, and Erik Wilde for the reviews and feedback.

9. References

9.1. Normative References

[I-D.ietf-core-coral]
Hartke, K., "The Constrained RESTful Application Language (CoRAL)", Work in Progress, Internet-Draft, draft-ietf-core-coral-03, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-core-coral-03.txt>.

[I-D.ietf-core-dev-urn]
Arkko, J., Jennings, C., and Z. Shelby, "Uniform Resource Names for Device Identifiers", Work in Progress, Internet-Draft, draft-ietf-core-dev-urn-04, 13 December 2019, <http://www.ietf.org/internet-drafts/draft-ietf-core-dev-urn-04.txt>.

[I-D.ietf-core-dynlink]
Shelby, Z., Koster, M., Groves, C., Zhu, J., and B. Silverajan, "Dynamic Resource Linking for Constrained RESTful Environments", Work in Progress, Internet-Draft, draft-ietf-core-dynlink-10, 22 July 2019, <http://www.ietf.org/internet-drafts/draft-ietf-core-dynlink-10.txt>.

[I-D.ietf-core-href]
Hartke, K., "Constrained Resource Identifiers", Work in Progress, Internet-Draft, draft-ietf-core-href-03, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-core-href-03.txt>.

[I-D.ietf-core-resource-directory]
Shelby, Z., Koster, M., Bormann, C., Stok, P., and C. Amsuess, "CoRE Resource Directory", Work in Progress, Internet-Draft, draft-ietf-core-resource-directory-24, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-core-resource-directory-24.txt>.

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", Ph.D. Dissertation, University of California, Irvine , 2000.

[RFC3986] Berners-Lee, T., Fielding, R., and L. Masinter, "Uniform Resource Identifier (URI): Generic Syntax", STD 66, RFC 3986, DOI 10.17487/RFC3986, January 2005, <https://www.rfc-editor.org/info/rfc3986>.

[RFC5246] Dierks, T. and E. Rescorla, "The Transport Layer Security (TLS) Protocol Version 1.2", RFC 5246, DOI 10.17487/RFC5246, August 2008, <https://www.rfc-editor.org/info/rfc5246>.

[RFC5280] Cooper, D., Santesson, S., Farrell, S., Boeyen, S., Housley, R., and W. Polk, "Internet X.509 Public Key Infrastructure Certificate and Certificate Revocation List (CRL) Profile", RFC 5280, DOI 10.17487/RFC5280, May 2008, <https://www.rfc-editor.org/info/rfc5280>.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, DOI 10.17487/RFC5988, October 2010, <https://www.rfc-editor.org/info/rfc5988>.

[RFC6202] Loreto, S., Saint-Andre, P., Salsano, S., and G. Wilkins, "Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP", RFC 6202, DOI 10.17487/RFC6202, April 2011, <https://www.rfc-editor.org/info/rfc6202>.

[RFC6347] Rescorla, E. and N. Modadugu, "Datagram Transport Layer Security Version 1.2", RFC 6347, DOI 10.17487/RFC6347, January 2012, <https://www.rfc-editor.org/info/rfc6347>.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, DOI 10.17487/RFC6690, August 2012, <https://www.rfc-editor.org/info/rfc6690>.

[RFC7049] Bormann, C. and P. Hoffman, "Concise Binary Object Representation (CBOR)", RFC 7049, DOI 10.17487/RFC7049, October 2013, <https://www.rfc-editor.org/info/rfc7049>.

[RFC7230] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Message Syntax and Routing", RFC 7230, DOI 10.17487/RFC7230, June 2014, <https://www.rfc-editor.org/info/rfc7230>.

[RFC7231] Fielding, R., Ed. and J. Reschke, Ed., "Hypertext Transfer Protocol (HTTP/1.1): Semantics and Content", RFC 7231, DOI 10.17487/RFC7231, June 2014, <https://www.rfc-editor.org/info/rfc7231>.

[RFC7641] Hartke, K., "Observing Resources in the Constrained Application Protocol (CoAP)", RFC 7641, DOI 10.17487/RFC7641, September 2015, <https://www.rfc-editor.org/info/rfc7641>.

[RFC7959] Bormann, C. and Z. Shelby, Ed., "Block-Wise Transfers in the Constrained Application Protocol (CoAP)", RFC 7959, DOI 10.17487/RFC7959, August 2016, <https://www.rfc-editor.org/info/rfc7959>.

[RFC8446] Rescorla, E., "The Transport Layer Security (TLS) Protocol Version 1.3", RFC 8446, DOI 10.17487/RFC8446, August 2018, <https://www.rfc-editor.org/info/rfc8446>.

[RFC8613] Selander, G., Mattsson, J., Palombini, F., and L. Seitz, "Object Security for Constrained RESTful Environments (OSCORE)", RFC 8613, DOI 10.17487/RFC8613, July 2019, <https://www.rfc-editor.org/info/rfc8613>.

[W3C.REC-exi-20110310]
Schneider, J. and T. Kamiya, "Efficient XML Interchange (EXI) Format 1.0", World Wide Web Consortium Recommendation REC-exi-20110310, 10 March 2011, <http://www.w3.org/TR/2011/REC-exi-20110310>.

[W3C.REC-html5-20141028]
Hickson, I., Berjon, R., Faulkner, S., Leithead, T., Navara, E., O&#039;Connor, T., and S. Pfeiffer, "HTML5", World Wide Web Consortium Recommendation REC-html5-20141028, 28 October 2014, <http://www.w3.org/TR/2014/REC-html5-20141028>.

9.2. Informative References

[CollectionJSON]
Amundsen, M., "Collection+JSON - Document Format", February 2013, <http://amundsen.com/media-types/collection/format/>.

[I-D.bormann-core-media-content-type-format]
Bormann, C., "On Media-Types, Content-Types, and related terminology", Work in Progress, Internet-Draft, draft-bormann-core-media-content-type-format-01, 8 July 2019, <http://www.ietf.org/internet-drafts/draft-bormann-core-media-content-type-format-01.txt>.

[I-D.handrews-json-schema-validation]
Wright, A., Andrews, H., and B. Hutton, "JSON Schema Validation: A Vocabulary for Structural Validation of JSON", Work in Progress, Internet-Draft, draft-handrews-json-schema-validation-02, 17 September 2019, <http://www.ietf.org/internet-drafts/draft-handrews-json-schema-validation-02.txt>.

[I-D.hartke-core-apps]
Hartke, K., "CoRE Applications", Work in Progress, Internet-Draft, draft-hartke-core-apps-08, 22 October 2018, <http://www.ietf.org/internet-drafts/draft-hartke-core-apps-08.txt>.

[I-D.ietf-ace-dtls-authorize]
Gerdes, S., Bergmann, O., Bormann, C., Selander, G., and L. Seitz, "Datagram Transport Layer Security (DTLS) Profile for Authentication and Authorization for Constrained Environments (ACE)", Work in Progress, Internet-Draft, draft-ietf-ace-dtls-authorize-09, 18 December 2019, <http://www.ietf.org/internet-drafts/draft-ietf-ace-dtls-authorize-09.txt>.

[I-D.ietf-ace-oauth-authz]
Seitz, L., Selander, G., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "Authentication and Authorization for Constrained Environments (ACE) using the OAuth 2.0 Framework (ACE-OAuth)", Work in Progress, Internet-Draft, draft-ietf-ace-oauth-authz-33, 6 February 2020, <http://www.ietf.org/internet-drafts/draft-ietf-ace-oauth-authz-33.txt>.

[I-D.ietf-ace-oscore-profile]
Palombini, F., Seitz, L., Selander, G., and M. Gunnarsson, "OSCORE profile of the Authentication and Authorization for Constrained Environments Framework", Work in Progress, Internet-Draft, draft-ietf-ace-oscore-profile-10, 9 March 2020, <http://www.ietf.org/internet-drafts/draft-ietf-ace-oscore-profile-10.txt>.

[I-D.ietf-core-coap-pubsub]
Koster, M., Keranen, A., and J. Jimenez, "Publish-Subscribe Broker for the Constrained Application Protocol (CoAP)", Work in Progress, Internet-Draft, draft-ietf-core-coap-pubsub-09, 30 September 2019, <http://www.ietf.org/internet-drafts/draft-ietf-core-coap-pubsub-09.txt>.

[IANA-CoAP-media]
"CoAP Content-Formats", n.d., <http://www.iana.org/assignments/core-parameters/core-parameters.xhtml#content-formats>.

[IANA-media-types]
"Media Types", n.d., <http://www.iana.org/assignments/media-types/media-types.xhtml>.

[RFC5789] Dusseault, L. and J. Snell, "PATCH Method for HTTP", RFC 5789, DOI 10.17487/RFC5789, March 2010, <https://www.rfc-editor.org/info/rfc5789>.

[RFC6763] Cheshire, S. and M. Krochmal, "DNS-Based Service Discovery", RFC 6763, DOI 10.17487/RFC6763, February 2013, <https://www.rfc-editor.org/info/rfc6763>.

[RFC6943] Thaler, D., Ed., "Issues in Identifier Comparison for Security Purposes", RFC 6943, DOI 10.17487/RFC6943, May 2013, <https://www.rfc-editor.org/info/rfc6943>.

[RFC7228] Bormann, C., Ersue, M., and A. Keranen, "Terminology for Constrained-Node Networks", RFC 7228, DOI 10.17487/RFC7228, May 2014, <https://www.rfc-editor.org/info/rfc7228>.

[RFC7252] Shelby, Z., Hartke, K., and C. Bormann, "The Constrained Application Protocol (CoAP)", RFC 7252, DOI 10.17487/RFC7252, June 2014, <https://www.rfc-editor.org/info/rfc7252>.

[RFC7320] Nottingham, M., "URI Design and Ownership", BCP 190, RFC 7320, DOI 10.17487/RFC7320, July 2014, <https://www.rfc-editor.org/info/rfc7320>.

[RFC7925] Tschofenig, H., Ed. and T. Fossati, "Transport Layer Security (TLS) / Datagram Transport Layer Security (DTLS) Profiles for the Internet of Things", RFC 7925, DOI 10.17487/RFC7925, July 2016, <https://www.rfc-editor.org/info/rfc7925>.

[RFC8075] Castellani, A., Loreto, S., Rahman, A., Fossati, T., and E. Dijk, "Guidelines for Mapping Implementations: HTTP to the Constrained Application Protocol (CoAP)", RFC 8075, DOI 10.17487/RFC8075, February 2017, <https://www.rfc-editor.org/info/rfc8075>.

[RFC8132] van der Stok, P., Bormann, C., and A. Sehgal, "PATCH and FETCH Methods for the Constrained Application Protocol (CoAP)", RFC 8132, DOI 10.17487/RFC8132, April 2017, <https://www.rfc-editor.org/info/rfc8132>.

[RFC8152] Schaad, J., "CBOR Object Signing and Encryption (COSE)", RFC 8152, DOI 10.17487/RFC8152, July 2017, <https://www.rfc-editor.org/info/rfc8152>.

[RFC8259] Bray, T., Ed., "The JavaScript Object Notation (JSON) Data Interchange Format", STD 90, RFC 8259, DOI 10.17487/RFC8259, December 2017, <https://www.rfc-editor.org/info/rfc8259>.

[RFC8392] Jones, M., Wahlstroem, E., Erdtman, S., and H. Tschofenig, "CBOR Web Token (CWT)", RFC 8392, DOI 10.17487/RFC8392, May 2018, <https://www.rfc-editor.org/info/rfc8392>.

[RFC8428] Jennings, C., Shelby, Z., Arkko, J., Keranen, A., and C. Bormann, "Sensor Measurement Lists (SenML)", RFC 8428, DOI 10.17487/RFC8428, August 2018, <https://www.rfc-editor.org/info/rfc8428>.

[RFC8576] Garcia-Morchon, O., Kumar, S., and M. Sethi, "Internet of Things (IoT) Security: State of the Art and Challenges", RFC 8576, DOI 10.17487/RFC8576, April 2019, <https://www.rfc-editor.org/info/rfc8576>.

[RFC8747] Jones, M., Seitz, L., Selander, G., Erdtman, S., and H. Tschofenig, "Proof-of-Possession Key Semantics for CBOR Web Tokens (CWTs)", RFC 8747, DOI 10.17487/RFC8747, March 2020, <https://www.rfc-editor.org/info/rfc8747>.

[W3C-TD] Kaebisch, S., Kamiya, T., McCool, M., Charpenay, V., and M. Kovatsch, "Web of Things (WoT) Thing Description", April 2020, <https://www.w3.org/TR/wot-thing-description/>.

Appendix A. Future Work

* Unreliable (best effort) communication, robust communication in network with high packet loss, 3-way commit

* Discuss directories, such as CoAP Resource Directory

* More information on how to design resources; choosing what is modeled as a resource, etc.

Authors' Addresses

Ari Keranen
Ericsson
FI-02420 Jorvas
Finland

Email: ari.keranen@ericsson.com

Matthias Kovatsch
Siemens AG
Otto-Hahn-Ring 6
D-81739 Munich
Germany

Email: matthias.kovatsch@siemens.com

Klaus Hartke
Ericsson
Torshamnsgatan 23
SE-16483 Stockholm
Sweden

Email: klaus.hartke@ericsson.com