Event-Driven Communication on Application Level in a Smart Home

Christoph Soellner, Uwe Baumgarten
Chair for Operating Systems (F13)
Technische Universitaet Muenchen
Munich, Germany
{cs, baumgaru}@tum.de

***Abstract*—In recent years, research on the Internet of Things focused on wired or wireless information transport. Progress has been made in energy-efficiency, reduction of bandwidth usage and bringing standard Internet protocols to small resource constrained devices. Yet, an application level protocol that not only aims at enabling such devices for remote control but at the same time also offers semantic description features to human users is not found. Based upon previous work, we introduce further semantic schema extensions, add event-based communication on application level and compare our approach to existing work.**

Keywords—Smart Home; Smart Grid; Smart Device; remote control; Internet of Things; embedded system

I. INTRODUCTION

In a previous work [1], we introduced a straightforward concept aimed at tiny embedded devices to enable them for remote control on application level via Internet standard protocols. We showed how to use the Extensible Markup Language (XML) [2] to describe devices and their capabilities. Furthermore, we used two methods (`GET` and `POST`) from the ReSTful hypertext transport protocol (HTTP) [3] to enable devices for basic remote control by standard HTTP clients.

We also described briefly our implementation that was written for the Atmel microcontroller unit (MCU) families *ATmega* [4] and *XMega* [5] with at least 16 kByte of program memory and 1 kByte of static RAM. The software features a state machine based HTTP parser written in *C* to process messages in a byte-sequential manner. Thus, it was intended be used over low-performance communication links that transmit one byte at a time only, such as a universal asynchronous receiver / transmitter (USART) link.

To also achieve a reduced memory footprint, the parser will process an HTTP message separated according to its components such as the method, the URL path, optionally its header information and body values. It does not store the message completely, but processes it on-the-fly instead.

In our concept we identified a few unresolved issues. First, numeric XML elements in some cases may not carry enough semantic information for a human user. Secondly, a Smart Home setup will require sensors and actuators to communicate upon certain conditions without a third party.

After recapitulating key elements of our concept in section II, we describe the introduction of additional XML elements to our schema and the extension of the control interface to also support event driven communication between Smart Devices in section III, and finally, compare our solution to existing protocols in section IV.

II. RELATED WORK

In our previous work [1], we proposed a hierarchical three-level XML based description of devices, where the first level offers device and meta information, the second level serves as a container to group machine state information semantically, and the third level carries said machine state information. We also introduced a ReSTful control interface, the commands of which are derived from a device's XML description.

Principal goals with our new approach were to show that already proven internet standard protocols can be processed on very resource constrained MCUs and to demonstrate the importance of communication layer separation. Within a Smart Home, IP based communication between all devices cannot be assumed; tasks such as addressing must be handled on application level alone, when only a broadcast message transport is available.

To demonstrate the protocol efficiency, we implemented our concept in *C* for Atmel AVR MCUs. By utilizing preprocessor macros, we ensure small code while at the same time providing comfortable configuration options to the developer. Modification of machinery state is done through callback functions which are implemented by the developer and registered with our library; they are called whenever the HTTP parser has determined a specific action from a received message.

An exemplary binary code, which utilizes our library, a USART communication driver and basic get/set functions consumes roughly 14 kByte of program memory space and thus fits into our target MCU family. It simulates a combined refrigerator / freezer device and serves as example in the subsequent sections, in which we will cover the ideas of the application level interface.

A. Service description

In general, we assume a Smart Device to be a black box and unique; there is no schema to a device, since, in principle, every vendor may choose to implement certain functionality in their own way. Therefore, a device must be queried for its capabilities, at least once before control operations are possible. Any Smart Device that implements our application interface and is queried for its features, will output an XML description as given in Fig 1. As mentioned, we operate solely on application level and cannot make any assumptions as to how the messages are relayed. In particular, we cannot assume IP based communication.

Furthermore, a principal concept is our integration of both semantic and machine-interpretable information into one single description, as depicted in Fig. 1.

 ISBN: 978-1-61208-282-0

```
<dev00010>
  <meta type="deviceName">TUM Refridgerator</meta>
  <meta type="manufacturerName">Chair F13, TUM</meta>
  <meta type="manufacturerURL">http://www.os.in.tum.de/</meta>
  <Fridge>
    <Door type="string" access="readonly">closed</Door>
    <CurrentTemperature type="float" access="readonly" ↵
     unit="°C">8.6</CurrentTemperature>
    <TargetTemperature type="float" access="readwrite" ↵
     min="4.0" max="16.0" unit="°C">8.0</TargetTemperature>
  </Fridge>
  <Freezer>
    <Door type="string" access="readonly">closed</Door>
    <CurrentTemperature type="float" access="readonly" ↵
     unit="°C">-17.6</CurrentTemperature>
    <TargetTemperature type="float" access="readwrite" ↵
     min="-30.0" max="-8.0" unit="°C">-18.0</TargetTemperature>
  </Freezer>
</dev00010>
```

Figure 1. XML discovery message of a Smart Device

We use one single XML document generated on-the-fly by the Smart Device to deliver both semantic information suited for a human user and machine state information intended for automated processing. Semantic information is given by naming XML elements appropriately, data relevant to machines is provided through a node's respective attributes and values. Thus, element names differ from device to device and are chosen appropriately according to generic XML node naming rules [6] by the developer. Node names are treated *case-sensitive*.

1) Device level

As denoted in the example, the first level node carries a unique id for a device (*dev00010*) and may further contain *meta* elements. Each of those consists of a designating *type* attribute and a corresponding *value*. It has, by definition, no meaning for machine-to-machine communication and is only presented to help a human user understand the device's purpose. Thus, it will be displayed unmodified to the user by the application. We allow and encourage simple interpretation such as displaying hyperlinks in a clickable manner.

2) Service level

The service level offers a way of displaying separate hardware parts within a Smart Device to a user. As shown in Fig. 1, both the refrigerator and the freezer can be described in a similar fashion; simply by naming the service nodes differently and semantically plausible, a human user is able to distinguish and interpret contained data more easily.

3) Data point level

The actual machine state information is contained in nodes on this level. Each *service* has at least one data point, where a data point represents a single value of one of three types: integer, floating point number and string. Each value will be derived from internal machine state by a callback function when queried and will be modified by a different callback function when written to.

Our library also features the definition of other attributes such as *min* and *max* boundaries for numerical data points, *access* restrictions to allow for readonly or writeonly data points and a *unit* string helping human users interpret a data point's value. When set, restrictions will automatically be enforced upon modification of data points before they are passed to the respective callback function.

```
>>> GET /dev00010/ HTTP/1.0
>>> GET / HTTP/1.0
    Host: dev00010
>>> GET /*/ HTTP/1.0
>>> GET / HTTP/1.0
    Host: *
```

Figure 2. A single or all devices are addressed via HTTP.

B. Control interface

In this subsection, we describe the ReSTful command interface for a Smart Device. The importance of a ReSTful interface for resource constrained devices has been discussed in [8]. Control messages follow the HTTP message specification. We allow `GET` to retrieve one or more data points and `POST` to modify machine states. Other verbs are not supported.

1) Device addressing

The message transportation layer is not required to feature device addressing on hardware or protocol level. Instead, addressing is done in software on application level with HTTP alone.

To address one single or all devices connected to the same medium, a regular HTTP request is used with the first part of the URI being the target device id or the asterisk, meaning 'any', as shown in Fig. 2.

```
>>> GET / HTTP/1.1
    Host: dev00010
    Host: anotherDevice0293
    Host: technicsStereo10023
```

Figure 3. Several devices are addressed in a multicast manner.

When communication with several devices in a single (application level) multicast request is required, we utilize HTTP's `Host:`-header to address those devices by their respective id. As allowed by the specification [3], the `Host:`-header is set several times in this case. Fig. 3 gives an example. Note that the host values could also be sent as comma separated values in one single line; however, this would increase parser complexity and is not supported by our code.

```
<?xml version="1.0">
<bus>
  <dev00010><!-- […] --></dev00010>
  <anotherDevice0293><!-- […] --></anotherDevice0293>
</bus>
```

Figure 4. The enclosing bus element is displayed.

Each well-formed XML document must have exactly one root element. Therefore, the first device to respond will output the XML header and an enclosing XML root element named `bus`, when more than one device is *addressed*, as shown in Fig. 4. It will be ignored in all requests to the devices; its purpose is merely to fulfill XML document specification.

2) Service discovery

Previous example requests already depicted the Smart Devices' service discovery procedure: The command `GET /` is used query all connected devices for their complete XML description, which includes any meta information and data point attributes. The resulting document is generated on-the-fly each time a discovery procedure is initiated. Note that reliability and security issues are handled on underlying levels according to application requirements.

 ISBN: 978-1-61208-282-0

```
>>> GET /dev00010/*/Door HTTP/1.1
<<< <dev00010>
      <Fridge><Door>closed</Door></Fridge>
      <Freezer><Door>closed</Door></Freezer>
    </dev00010>
```

Figure 5. Machine state information is retrieved by specifying an XPath selector in the URL path.

3) Retrieving data point values

We utilize a reduced XPath [7] implementation to retrieve machine state stored in data point values. To that end, the URL path is used as XPath selector of the virtual document of the discovery output of devices. The asterisk character may be used on any of the three hierarchical levels to select all instances on that level, as depicted in Fig. 5.

As shown, no *meta* information and no attributes are output when not in discovery mode; it is assumed that a client already completed the discovery procedure and stored its result for subsequent requests. Thus, a retransmission serves no purpose.

4) Modifying data point values

The HTTP method `POST` is used to modify machine state.

```
>>> POST /dev00010/Freezer/TargetTemperature HTTP/1.1

    -18.5
```

Figure 6. A single value is modified through a POST request.

To set a single data point on a single Smart Device, the URL can be used in the same manner as in II.B.3). The value to be written is transmitted in the message's body (Fig. 6).

```
>>> POST /dev00010/ HTTP/1.1

    Fridge.TargetTemperature=6.0&Freezer.TargetTemperature=-18.5

>>> POST /*/ HTTP/1.1

    *.TargetTemperature=0.0
```

Figure 7. Several values are changed in a single message.

In order to achieve application level manycast, several or all values within a single device or even several devices can be changed through one single broadcast message, as depicted in Fig. 7.

Note that the separating character in the HTTP body is the decimal instead the forward slash. Thus, we achieve compatibility with hypertext markup language (HTML) forms and can offer a simple web interface that is automatically generated from the service and data point definition.

III. CONCEPT EXTENSION

In our research, we came across certain use cases where the current schema does not suffice. In this section, we extend the description schema and the protocol.

A. Data point labels

Numeric data points can not only be used to represent sensor readouts, they may also represent discreet machine states. A door for instance may either be locked or unlocked; instead of using a string representation, in this case a Boolean value suffices.

In general, a numeric data point with its name alone may not offer sufficient semantic information to a human user for interpretation. We give examples within a Smart Home:

- A window handle sensor may for instance only take on values of *0*, *1* and *2*; where one could guess a semantic meaning for *0* (*closed*) it would be feasible to describe *1* as *flipped* and *2* as *open* during discovery.
- A laundry machine that features a static program list may encode the currently selected program only in a numeric value. However, also a human interpretable text representation is required for a consumer.
- A Smart Radio receiver may output a station list derived from an auto tune process or decoded radio data system messages to be displayed in a remote control application.

```
<radio00295>
  <meta type="deviceName">Smart Radio</meta>
  <Tuner>
    <Frequency type="float" access="readwrite" ↵
     min="87.5" max="108.0" unit="MHz">
      96.3
      <Label from="96.3" >Bell 96,3</Label>
      <Label from="97.5" >Antenna Bavaria</Label>
      <Label from="107.2">Antenna Bavaria</Label>
    </Frequency>
  </Tuner>
</radio00295>
```

Figure 8. The Label element with its attributes "from" and "to" is introduced.

We therefore introduce the `<Label>` element for data point elements with two attributes, describing a label's valid range: *from* and *to*. The attributes' restriction base is equal to the parenting data point's data type, i.e., *from* and *to* may carry floating point values only if the parent data point is of the type *float*. In case a label is only valid for a single data point value and thus both attributes would show the same value, *to* may be omitted (Fig. 8).

```
<sensor20912>
  <Brightness>
    <Value type="int" access="readonly" ↵
     min="0" max="120000" unit="lx">
      1050
      <Label from="0" to="2000">darkness</Label>
      <Label from="60000" to="120000">daylight</Label>
      <Label from="80000" to="90000" >bright sunlight</Label>
      <Label from="90001" to="120000">bright sunlight</Label>
    </Value>
  </Brightness>
</radio00295>
```

Figure 9. Overlapping ranges for Label elements are displayed.

To specify a range, *to* must be given. Both boundaries can also take on `INF` or `-INF` within *float* data points. Ranges of different labels may overlap (Fig. 9). In this case, the client presents all suitable Label element values for a given data point value to the user. Label values that are valid for more than one data point value are given as many times as required (third Label element servers as demonstration).

Note that the specified element has no impact on the HTTP parser complexity on the MCU; it is output in discovery mode only.

 ISBN: 978-1-61208-282-0

B. Event initiated communication

Our concept, so far, only supports communication for retrieval or modification of machine state initiated by control software through `GET` or `POST`. We extend it to also support event driven notifications sent by Smart Devices upon certain conditions. To signal an event, we introduce a new HTTP verb and define the message format in the following subsections.

1) HTTP message format

An event message can be sent for any data point. All event messages will be broadcast by the Smart Device generating it, thus being received by all other Smart Devices on the same logical message transport segment. Also, it may be forwarded into other networks by different protocols such as UDP/IP.

```
>>> EVENT /dev00010/Fridge/Door HTTP/1.1
>>>
>>> open
```

Figure 10. An exemplary Event message by the Smart Fridge device is shown.

Event messages have the same structure as HTTP messages; the verb designates an operation concerning a specified resource designated by the URL path and the body contains the new value.

To that end, we introduce the method `EVENT`. As opposed to regular HTTP messages and in compliance with our previous statement however, the URI path designates the path to the *originating* (and *not* the destination) data point as shown in Fig. 10.

Furthermore, each `EVENT` message carries exactly one data point value in its body. In case there are several changed data points, the respective amount of messages is required. As already mentioned, we rely on lower layers to implement reliability and security.

The advantage of keeping event notifications in an HTTP format and allowing only singular values within the body is, that only little modification to the HTTP parser is necessary. We still do not require XML parsing capabilities, but can rather utilize already present code to process an event.

Since there is no publish/subscribe mechanism with our concept, `EVENT` messages are not being replied to.

2) Implementation overview

To achieve described functionality we extend the HTTP parser: in case of an `EVENT` notification, a hash table stored on the device is queried to find a suitable mapping of the event's URL path and an internal data point address. The hash table stores every such path as a mapping between a zero-terminated character string and an unsigned integer containing the program memory address of the respective data point.

If a mapping is found at least once, the event's body value is passed to the registered callback function for each such match. Restrictions and other data type mapping apply in the same way as with a regular `POST` message.

Event notifications usually do not generate a response. However, when a data point is modified through an event, it *may* be configured to in turn broadcast an event to confirm the change. This is particularly useful for monitoring machine state: a light for instance may have been turned on by a hardware button, an event message or a `POST` request – in all three cases events may be generated and could displayed on a monitoring instance, regardless of the source of the modification.

We make no assumption as to when an event message needs to be sent; it ultimately depends on the application. The Smart Light controller for instance may send events for each light affected as soon as there is no change for a certain period of time; in this case, dimming the light would not result in event message flooding.

For efficient implementation we recommend placing the hash table into the MCU's EEPROM with a static limit on its size. Thus, a Smart Device may offer only a limited number of such direct connections with other Smart Devices.

We are aware that ReSTful interfaces were designed for request / response operation only and that it may seem inappropriate to encode *origination information* in a resource locator at first. However, ReST can also be viewed as a concept where the *method* operates on the *resource specified*, with no actual locations or addresses given.

```
<smartLight>
  <Bulb>
    <State type="int" access="readwrite" min="0" max="1">1</State>
    <Toggle type="int" access="writeonly" min="1" max="1" />
  </Bulb>
</smartLight>
```

Figure 11. An exemplary service description for a Smart Light controller is displayed.

3) Example

Consider a Smart Light controller that features two data points (Fig. 11). The first one, "State", turns the light on and off, according to the values 1 and 0 written to it, respectively. The second one, "Toggle", negates the current switch's state in case a "1" is written to it.

Secondly, consider a Smart Switch ("sw2013") with a single button (service "Button"), which in turn contains one data point names "Pressed". It broadcasts an Event on two occasions: a `1` when pressed, and a `0` when released.

We configure the `Toggle` data point on the light controller to accept event messages from `/sw2013/Button/Pressed`. Since `Toggle` only accepts a `1` value, all *release* events are ignored. Therefore, this configuration results in push button functionality.

IV. COMPARISON WITH OTHER PROTOCOLS

In this section, we compare our approach with existing application level protocols with regards to functionality, features, message size and –if possible– implementation details. Note that no assumptions over message transport or underlying protocols can be made, i.e., the Internet Protocol in particular and other well-known protocols like TCP / UDP or the DNS are not available.

A. XML protocols

There are already some protocols available to be used within a Smart Home scenario, mostly based on XML. Yazar and Dunkels compared a ReSTful control interface with a SOAP [9] based approach; they found SOAP to have a significantly larger memory footprint, execution time and message size [10].

 ISBN: 978-1-61208-282-0

Also, SOAP based protocols rely on a service schema description written in the Web Service Description Language (WSDL) which must be stored and processed separately.

Even with a limited subset of SOAP functionality, e.g., the devices profile for web services [11] (DPWS), we are restricted to SAX like parsing on the target embedded hardware, since a request may not fit into the device's memory as a whole. Thus, essential features such as name spaces, message integrity checks or verification against a given WSDL schema may not be available and limit compatibility with standard clients.

Lastly, SOAP makes heavy use of DNS and IP for message routing. As stated, those may not be available on a Smart Device network. Therefore, a SOAP based approach for message exchange seems not feasible for the Smart Home Scenario.

B. Constrained Application Protocol

Mainly designed as a binary replacement for HTTP, the constrained application protocol [12] (CoAP) aims at enabling resource constrained embedded devices for internet communications. It introduces application level reliability, offers simplified parsing through binary header representation instead of a text based one and remains largely compatible with HTTP through standardized command mappings.

CoAP achieves a reduced message size compared to HTTP. We can easily confirm this by counting octets, e.g., for the message in Fig. 6: while HTTP results in 58 octets, we achieve a significantly smaller size for the CoAP equivalent, 47 octets (4 header, URI-Path 1+8, URI-Path 1+8, URI-Path 2+17, 1 octet separator, 5 octets payload).

However, major disadvantages, as we noted in [1], include the limitation to only four methods, the duplication of TCP's features of reliability on application level (where this should remain on the transport layer) and the tight integration with IP/UDP respectively. Also, the code size of a commonly used reference implementation is about ~25 kByte of program memory space, and additionally, CoAP does not cover service description or discovery procedures.

C. Constrained restful environments link format

To overcome the latter, Shelby et al. [14] also proposed application layer schemata for service consumption, service directories and caches [13], the Constrained ReSTful Environments link format (CoRE). While in principle the CoRE can be run over any ReSTful interface, it was designed to specifically work over CoAP, since it relies on CoAP's request and response code mapping and requires it to handle all machine addressing tasks.

In the current version of the RFC, we can identify weaknesses in the text based protocol:

- Use of angle brackets ("<>"): Both characters are used in markup languages to denote descriptive information. Within CoRE, an actual resource location is designated between those characters, which may confuse both designers and restrictive firewall software; moreover, as the forward slash ("/") is explicitly allowed.
- Discovery entry point: It is not clear why a `/.well-known/core` URI is used for discovery. Common web browsers, for instance, use the much shorter GET command `GET /`.
- Parser complexity: While CoAP was designed specifically to reduce parser complexity on application level, this is negated with the CoRE approach. According to [14], complex query search and filtering tasks can optionally be supported by Smart Devices with query filtering. All optional parts introduce uncertainty and unreliability into a concept; moreover, it is not evaluated which filtering options are required in the first place on resource constrained devices.

Although CoAP is routed over UDP links and supports delayed responses, neither CoRE nor CoAP specify event based broadcast messages. A Smart Device would therefore always have to have routing and IP address information about desired recipients. Particularly in scenarios with dynamic IP addresses (met constantly within a Smart Home), CoAP message links may become cumbersome to maintain.

While we acknowledge that Smart Home functionality, i.e., retrieving and modifying machinery state, can be achieved by a CoRE protocol design, the authors fail to clarify the advantages of their concept over regular XML and XPATH expressions; a reference implementation enabling a comparison of both approaches under similar circumstances is not yet available.

Lastly, both CoAP and CoRE have a strong disadvantage in common: They both need to be implemented again for each programming language and type of network equipment (such as firewalls and application level gateways) that is to feature the protocol. This is not necessary with HTTP and XML.

V. Conclusion and Oulook

With the extension of our concept targeted at tiny embedded Smart Devices, we resolved the issues occurring with some use cases on application level.

Furthermore, with the introduction of the HTTP method `EVENT` we demonstrated how Smart Devices can initiate communication and how this approach can be leveraged to enable vendor-independent application level communication between Smart Devices without application level translation.

In future work, we address repeated transmission of event messages in case a certain condition persists and demonstrate an application level message routing software that allows third parties, such as a power utility, to influence machine behavior on occasion (e.g., emergencies or power overproduction).

Also, we evaluate our implementation against CoRE in more detail, once a properly maintained implementation becomes available.

References

[1] C. Soellner and U. Baumgarten, "Bridging the Last Mile in a Smart Home on Application Level", The 4th Int. Conf. on Smart Communications in Network Technologies (SaCoNeT2013), in press.

[2] E. Ray, "Learning XML", 2nd ed., O'Reilly, 2003.

[3] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext Transport Protocol", Internet RFC 2616, Sep. 2004, Available: http://www.w3.org/Protocols/rfc2616/rfc2616-sec4.html#sec4.2, [retrieved April, 2013].

 ISBN: 978-1-61208-282-0

[4] Atmel vendor website, “megaAVR Microcontroller”, Available: http://www.atmel.com/products/microcontrollers/avr/megaavr.aspx, [retrieved May, 2013].

[5] Atmel vendor website, “AVR XMEGA Microcontroller”, Available: http://www.atmel.com/products/microcontrollers/avr/avr_xmega.aspx, [retrieved May, 2013].

[6] J. E. Simpson, “O’Reilly xml.com - The Naming of Parts”, Available: http://www.xml.com/pub/a/2001/07/25/namingparts.html, [retrieved April, 2013].

[7] M. Kay, “XPath 2.0 programmer's reference”, Indianapolis, IN: Wiley, 2004.

[8] D. Uckelmann, M. Harrison, and F. Michahelles, “Architecting the Internet of Things”, Springer-Verlag, Berlin, 2011, chapter 5.

[9] M. Gudgin et al., “SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)”, W3C Recommendation, Available: http://www.w3.org/TR/soap12-part1/, [retrieved May, 2013].

[10] D. Yazar and A. Dunkels, “Effcient application integration in IP-based sensor networks”, In Proceedings of the First ACM Workshop on Embedded Sensing Systems for Energy-Effciency in Buildings, BuildSys'09, pages 43-48, New York, NY, USA, 2009. ACM.

[11] T. Nixon and A. Regnier, “Devices Profile for Web Services (DPWS)”, OASIS Standard, Available: http://docs.oasis-open.org/ws-dd/ns/dpws/2009/01, [retrieved April, 2013].

[12] Z. Shelby, K. Hartke, and C. Bormann, “Constrained Application Protocol (CoAP)”, Internet-Draft 16 (work in progress), Available: http://tools.ietf.org/html/draft-ietf-core-coap-16, [retrieved May, 2013].

[13] Z. Shelby, S. Krco and C. Bormann, “CoRE Resource Directory”, Internet-Draft 5, Available: http://www.ietf.org/id/draft-shelby-core-resource-directory-05.txt, [retrieved May, 2013].

[14] Z. Shelby, “Constrained RESTful Environments (CoRE) Link Format”, IETF RFC 6690, Available: http://www.rfc-editor.org/rfc/rfc6690.txt, [retrieved May, 2013].

 ISBN: 978-1-61208-282-0