Observing Resources in CoAP
draft-hartke-coap-observe-01

Abstract

The state of a resource can change over time. We want to give clients of the CoRE WG CoAP protocol the ability to observe this change. This short I-D provides an example design for such an addition to CoAP, in order to be able to discuss the design alternatives in specific terms.

Status of this Memo

This Internet-Draft is submitted in full conformance with the provisions of BCP 78 and BCP 79.

Internet-Drafts are working documents of the Internet Engineering Task Force (IETF). Note that other groups may also distribute working documents as Internet-Drafts. The list of current Internet-Drafts is at http://datatracker.ietf.org/drafts/current/.

Internet-Drafts are draft documents valid for a maximum of six months and may be updated, replaced, or obsoleted by other documents at any time. It is inappropriate to use Internet-Drafts as reference material or to cite them other than as "work in progress."

This Internet-Draft will expire on January 9, 2011.

Copyright Notice

Copyright (c) 2010 IETF Trust and the persons identified as the document authors. All rights reserved.

This document is subject to BCP 78 and the IETF Trust’s Legal Provisions Relating to IETF Documents (http://trustee.ietf.org/license-info) in effect on the date of publication of this document. Please review these documents carefully, as they describe your rights and restrictions with respect to this document. Code Components extracted from this document must include Simplified BSD License text as described in Section 4.e of the Trust Legal Provisions and are provided without warranty as described in the Simplified BSD License.
Table of Contents

1.  Introduction ................................................. 3
2.  Subscription-lifetime Option ............................... 4
   2.1.  Example .................................................. 5
3.  Caching ..................................................... 6
4.  HTTP Mapping ................................................ 7
   4.1.  Example .................................................. 8
5.  IANA Considerations ........................................ 10
6.  Acknowledgments .............................................. 11
7.  References .................................................. 12
   7.1.  Normative References .................................... 12
   7.2.  Informative References ................................... 12
Appendix A.  Rationale ........................................... 13
   A.1.  Design Pattern .......................................... 13
   A.2.  Architecture ............................................ 13
   A.3.  Requirements ............................................ 14
   A.4.  The Design ............................................... 18
      A.4.1.  State ................................................ 18
      A.4.2.  Subscription Lifetime ............................... 18
      A.4.3.  Messages ............................................ 19
   A.5.  Message Exchanges ....................................... 20
      A.5.1.  Subscribing to a resource .......................... 21
      A.5.2.  Notifying of state changes ........................ 22
      A.5.3.  Unsubsctribing from a resource ..................... 23
      A.5.4.  Retrieving resource state .......................... 24
      A.5.5.  Changing resource state ............................. 25
      A.5.6.  Deleting a resource ................................ 25
   A.6.  Caching .................................................. 26
   A.7.  Identifying notifications and subscriptions .......... 26
   A.8.  Open issues ............................................. 28
Authors’ Addresses .............................................. 29
1. Introduction

The state of a resource can change over time. We want to give CoAP [I-D.ietf-core-coap] clients the ability to observe this change.

This short I-D provides an example design for such an addition to CoAP, in order to be able to discuss the design alternatives in specific terms.

In this document, the key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" are to be interpreted as described in BCP 14 [RFC2119] and indicate requirement levels for compliant CoAP implementations.
2. Subscription-lifetime Option

The Subscription-lifetime Option, when present, modifies the GET method to not only retrieve a representation of the current state of the resource identified by the request URI once, but also to notify the client of changes to the resource state for the duration specified in the option.

The client is notified of state changes by additional responses sent from the server to the client (confirmable or non-confirmable). The server MUST include the request URI, token and remaining subscription lifetime in each such notification response.

The subscription ends when the server sends a response with a 4xx or 5xx code, the client sends a RST in reply to a confirmable response, or the subscription lifetime expires.

It is not necessary that a subscribed client receives every single notification response sent by a server, or that the server sends a notification response for every single state change. However, the state observed by an observer SHOULD eventually become consistent with the actual state of the observed resource.

The representation format (i.e. the media type) used during the lifetime of a subscription MUST NOT change. If the server is unable to continue sending notification responses to a client in the requested representation format, it MUST send a confirmable 406 response and end the subscription.
2.1. Example

Client | Server
--------|--------
CON tid=47
GET /foo
lifetime=60s

ACK tid=47
200 /foo
"<temp... 

(The URI is actually elided as it is implied by the TID.)

... Time Passes ...

NON tid=153
200 /foo
"<temp...

Here, the server decides it will send updates often enough that a non-confirmable message is sufficient.

... Time Passes ...

CON tid=783
200 /foo
"<temp...

Here, the server decides to send a confirmable message as the time to the next update is not certain yet.

ACK tid=783

... and so on...

Figure 1
3. Caching

A client may cache part or all of a resource’s state changes. This allows clients to make conditional requests: If a subscribing client has cached a particular resource state, the server offering the resource does not need to send a full notification response.

- A client informs the server of its cached resource states by including Etags of cached resource states in the subscription request. This can be multiple Etags, as this enables the server to just send a "304 not modified" for a different Etag instead of a different representation if the resource switches around between multiple states. (Note that this list might be updated by a subscription refresh.)

- A server notifies a client of a change to a cached state by omitting the representation of the resource state in the notification and sets the code to "304 not modified" to indicate that the state changed to a cached state.

Another approach to caching multiple values for a resource is to express each possible value of the resource’s representation by a reference to another (unchanging) resource. This lets a resource change states between a set of such references that then provide the actual state information. A subscribed client fetches the individual resources on demand and caches the results for future use.
4. HTTP Mapping

Making a CoAP GET request with Subscription-lifetime Option available through an HTTP proxy requires techniques that allow an HTTP server (here: the proxy) to push data to an HTTP client, such as long polling or HTML5 WebSockets.

With long polling, the client requests information from the proxy in the same way as a normal HTTP GET request, but also indicates the last cached state of the requested resource. As long as the resource state hasn’t changed, the proxy holds the request and waits for the resource to change, instead of sending an empty (304) response. Once the state of the resource changes (or possibly after a suitable timeout), a complete response is sent to the client. The client will normally then immediately re-request information from the server, so that the client will generally be quickly notified of any change to the state of the resource.
4.1. Example

HTTP              CoAP

Client             Proxy             Server

<table>
<thead>
<tr>
<th>GET coap://..</th>
<th>CON tid=47</th>
<th>Setting up the initial state</th>
</tr>
</thead>
<tbody>
<tr>
<td>GET /foo</td>
<td>GET /foo</td>
<td>200 &quot;&lt;temp...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etag=ab5f78</td>
</tr>
<tr>
<td></td>
<td></td>
<td>ACK tid=47</td>
</tr>
<tr>
<td></td>
<td></td>
<td>200 &quot;&lt;temp...</td>
</tr>
<tr>
<td></td>
<td></td>
<td>etag=0xab5f78</td>
</tr>
</tbody>
</table>
|                 |                         | <--------------------------+
| GET coap://..   | CON tid=153             | Obtaining asynchronous      |
|                 |                         | updates                     |
| ?wmf=ab5f78     | GET /foo                | 304 "Not...                |
| &for=60s        |                         | etag=0xab5f78               |
|                 |                         | lifetime=60s                |
|                 |                         | ACK tid=153                 |
|                 |                         | 304 "Not...                |
|                 |                         | etag=0xab5f78               |
|                 |                         | <--------------------------+
|                 |                         | ... Time Passes ...         |
| 200 "<temp...   | CON tid=783             | HTTP client                 |
| etag=7b8fbb     |                         | will usually                |
|                 |                         | re-request here             |
|                 |                         | 200 "<temp...              |
|                 |                         | etag=0x7b8fbb               |
|                 |                         | ACK tid=783                 |
|                 |                         | +--------------------------+

Figure 2

In Figure 2, the HTTP client indicates the Etag that reflects its current state in a query parameter ("wmf" = When-Modified-From) and specifies a time in which it would like to receive an answer in any case ("for"). In this specific example, a change actually arrives within the time specified, so the HTTP client receives a 200 response to the long-poll. (If the time specified were to elapse without a change, a 304 response would be returned.) Figure 2 is not showing that the HTTP client most likely immediate re-requests the long-poll GET with the "wmf" parameter set to the new Etag.
Instead of specifying the parameters "wmf" and "for" in query parameters, the lifetime could also be implied by the proxy (e.g., the proxy could always set it to 60 seconds), and the Etag specifying the current state could also be transported in a new HTTP header such as "When-Modified-From:"

This example shows just one possible HTTP mapping, using a style of long-polls as they are often used for asynchronous HTTP applications [S6]. Alternative mappings are conceivable using HTML5 WebSockets or even other forms of asynchronous notifications via HTTP.
5. IANA Considerations

This draft adds the following option numbers to Table 2 of [I-D.ietf-core-coap]:

<table>
<thead>
<tr>
<th>Type</th>
<th>C/E</th>
<th>Name</th>
<th>Data type</th>
<th>Length</th>
<th>Default</th>
</tr>
</thead>
<tbody>
<tr>
<td>10</td>
<td>E</td>
<td>Subscription-lifetime</td>
<td>Duration</td>
<td>1 B</td>
<td>0</td>
</tr>
</tbody>
</table>

(The representation of the duration is assumed to be based on the Duration data type defined in [I-D.bormann-coap-misc]; alternatively it could be a variable-length integer specifying the duration in seconds.)
6. Acknowledgments

This work was partially funded by the Klaus Tschira Foundation.
7. References

7.1. Normative References

[I-D.bormann-coap-misc]

[I-D.ietf-core-coap]


7.2. Informative References


(Seminal dissertation introducing the REST architectural style.)


(S6 synchronized slide show system based on HTTP long-polls.)
Appendix A. Rationale

This appendix documents the considerations that led to the design specified in the main body of this document.

A.1. Design Pattern

Many designs are possible for the observe capability of CoAP. So that we don’t end up with a random, arbitrary design, we base our considerations on the well-known subject/observer design pattern. In this pattern, an object, called the subject, maintains a list of interested parties, called observers, and notifies them automatically of any state changes.

There are a number of variants of that design pattern. We like one that explicitly considers the way the evolution of the resource state might end [DUAL]. In detail, this variant of the design pattern consists of the following elements:

- A _subject_, which sends notifications to observers. It has a single method, SUBSCRIBE, which is called by observers that wish to receive notifications from the subject.

- An _observer_, which receives notifications from a subject. It has three methods: YIELD, which supplies the observer with new or current information; THROW, which informs the observer that the subject experienced an error condition; and BREAK, which indicates that the subject has finished sending notifications. The grammar of notifications to be expected over time therefore is:

  \[ \text{YIELD}^* ( \text{BREAK} | \text{THROW} )? \]

- A _subscription_, which represents the interest of an observer in a subject. It has a single method, UNSUBSCRIBE, which enables the subject to unsubscribe observers when notification has finished. Observers receive a reference to the subscription from the SUBSCRIBE method, so they can also call the UNSUBSCRIBE method to unsubscribe before the subject has finished sending notifications.

In this document, we describe an architecture and a protocol design that realizes a meaningful rendition of this design pattern within the REST-based [REST] environment of CoAP.

A.2. Architecture

We interpret resources as the _subjects_ of the subject/observer pattern. The _subscription_ causes the subject to continuously supply an _observer_ with the state of the resource: once upon
subscription and then whenever the state of the resource changes. We
call a CoAP node offering a resource _server_, and a CoAP node
subscribing an observer to a resource _client_. As with the existing
REST methods, this architecture is about exchanging representations
of resources, not about the messages (or method calls).

The design is REST-based, as it maintains REST’s uniform interface
and its four interface constraints ([REST], section 5.1.5):

- Identification of resources by the uniform mechanism of URIs
  ([RFC3986]);
- Resource manipulation through the exchange of resource state
  representations;
- Self-describing messages (potentially with multiple representation
  formats);
- Hypermedia as the engine of application state: A server
  premediates application state transitions by providing links in
  resources.;

While this is not a necessary implication of the design described
here, the assumption is that the subscription itself is modeled on
the level of conversation state, not as a resource by itself. (This,
of course, does not imply that there can’t be resources that help
nodes decide whether to create subscriptions, but these are
application-specific and outside the scope of CoAP.)

A.3. Requirements

The requirements for implementing the subject/observer design pattern
over UDP stem largely from the fact that UDP is an unreliable,
connectionless transport. This means that method calls must be
expressed as messages, that preparation must be taken for the case
that messages arrive out of order, appear duplicated, or go missing
without notice, and that the transport keeps no state between
messages that can be utilized.

The detailed requirements that follow from this are:

- An invocation of the SUBSCRIBE method on an observable resource is
  implemented by sending a message (a subscription request) from the
  subscribing client to the server that offers the resource.

  The client must be able to determine if a subscription request was
  received by the server, and, if not, must be able to retransmit
  the request.
The server must acknowledge the subscription request, and must be prepared to receive duplicated subscription requests.

Since subscribing can be made idempotent (Appendix A.7), the server need not be able to detect a duplicated subscription request as such.

The client must be able to relate the acknowledgment to the subscription request.

- The usual considerations for retrieving the representation of a resource in a REST-based protocol apply, e.g.:

  A subscribing client must be able to influence the representation format in which the server supplies the resource state to the client.

- Upon subscription, an observer must be supplied with the current state of the resource in the requested format.

  If the observer cannot be supplied with the current state (for example, because the resource does not exist, the state cannot be represented in the requested format, the client lacks authorization, or a general error condition occurred), the server must inform the client of the error condition.

  For efficiency, this initial notification may be sent within the same datagram that acknowledges the subscription request.

  The initial notification might not include the actual representation (e.g., take the form of a "304 not modified") if the client already has a valid representation of the resource. The client should be able to supply information (e.g., Etags or a modification date) to enable the server to make this determination.

- To take advantage of the multicast capabilities of the transport, it should be possible to subscribe a UDP multicast group to a resource. In contrast to subscribing multiple clients individually to the resource, the server in this case must treat the multicast group as a single observer.

- For robustness, a subscription has to be maintained through periodic refreshing. If a subscription is not refreshed, its lifetime must end after a certain duration that is negotiated as part of the message exchange that implements the SUBSCRIBE method call.
So a subscribing client must be able to specify a subscription lifetime duration in a subscription request. A server must be able to return the negotiated subscription lifetime duration back to the client.

Since the client is responsible for taking care of the subscription, refreshing a subscription must be implemented by sending a message (a subscription refresh request) from the subscribed client to the server.

The client must be able to determine if a subscription refresh request was received by the server, and, if not, must be able to retransmit the request.

The server must acknowledge the request. It must be prepared to receive duplicated requests. It must be able to relate a refresh request to a subscription.

The client must be able to relate the acknowledgment to the subscription refresh request.

The server must be prepared for a refresh request to arrive after the subscription expired. In this case, the subscription refresh request is treated the same as a subscription request, since the client expressed the desire to continue being subscribed to the resource.

For notifications, the equivalent of an invocation of the YIELD, BREAK or THROW method on an observer must be implemented by sending a notification message from the server offering the resource to the subscribed client.

The subscribed client must be able to relate a notification message to a subscription and/or to a resource (see also Appendix A.7).

It is not a requirement that a subscribed client receives every single notification messages sent by a server, or that the server sends a notification message for every single state change. However, the state observed by an observer must eventually become consistent with the actual state of the observed resource.

The representation format used during the lifetime of a subscription must not change. If the server is unable to continue notifying a client in the requested representation format, it must invoke the THROW method on the observer.
o A server must not send any further notification messages after sending a notification message that denotes a BREAK or THROW.

However, a client must be prepared to receive notification messages after receiving such a notification message. (In this case, it discards the excessive notification messages.)

o For robustness, a server can request the acknowledgment of a notification message from a client. (For example, in order to check if the client is still there, or to make sure that an observer observes a particular resource state.) Such a notification is called a _confirmable_ notification message.

A server must be able to determine if a confirmable notification message was received by the client, and, if not, must retransmit the message.

If the client cannot relate the confirmable notification message to a subscription, it must reject the message. Otherwise, it must acknowledge the message.

The server must be able to relate the acknowledgment or rejection to the confirmable notification message.

There is no support for multicasting confirmable notification messages.

o To end a subscription before the lifetime of the subscription expires (UNSUBSCRIBE method), a message (an ‘unsubscribe’ request) can be sent from the subscribed client to the server that maintains the subscription.

The client must be able to determine if an ‘unsubscribe’ request was indeed received by the server, and, if not, must be able to retransmit the request.

The server must acknowledge the ‘unsubscribe’ request, and must be prepared to receive duplicated requests (which also need to be acknowledged).

The client must be able to relate the acknowledgment to the request.

Note that, in a REST-based environment, all these message exchanges should also work correctly through proxies.
A.4. The Design

A.4.1. State

The state required on nodes as indicated by the requirements, can be summarized as follows:

- Nodes that send requests must keep track of pending requests. (For non-idempotent requests, the node that receives the request also must retain state to deduplicate requests.)

- A server offering a resource must keep track of the observers of the resource. Similarly, a proxy must keep track of the observers that observe a resource through that proxy.

- Due to the initial notification of an observer upon subscription, the server must also keep track the resource state itself. (It cannot just notify all observers when the state changes and then immediately forget the state.)

- A client subscribed to a resource may have to keep track of the subscriptions to the resource in order to be able to relate notifications to the subscription and in order to be able to refresh the subscription before the subscription lifetime ends (Appendix A.7).

- A client (or a proxy) may optionally keep a cache of resource states.

A.4.2. Subscription Lifetime

To summarize, the lifetime of a subscription begins with a subscription request, and it ends when

- the subscription lifetime expires (as defined by the subscription duration option in the subscription request),

- the client unsubscribes from the resource,

- the client rejects a request related to a subscription,

- an error condition related to a subscription occurred, or

- the observed resource has finished sending notifications.

A client can extend the lifetime of a subscription before its end by sending a subscription refresh request.
A.4.3. Messages

The following message types and elements can be gathered from the requirements.

A.4.3.1. Requests

Requests are messages that need to be acknowledged by the recipient. As with other request in CoAP, they are retransmitted by the sender using an exponential back-off delay until the acknowledgment is received.

The request message types as indicated by the requirements can be summarized as follows:

- A SUBSCRIBE request creates a new subscription or refreshes an existing subscription.
- An UNSUBSCRIBE request ends an existing subscription.

To put this into perspective, the existing GET, PUT, POST and DELETE messages have request semantics as well.

- A GET request retrieves a representation of the current resource state.
- A PUT request provides a new resource state in some representation format.
- A POST request creates or extends a resource.
- A DELETE request deletes a resource.

A request message includes a _transaction identifier_ which allows the recipient to detect duplicated requests and, by inclusion of the transaction identifier in the acknowledgment to the request, enables the sender to relate the acknowledgment to the request.

A.4.3.2. Replies

Replies are messages that are sent in reply to a request. They carry the aforementioned transaction identifier.

The reply messages types as indicated by the requirements can be summarized as follows:
o An ACK reply indicates an acknowledgment of a request.

o A RST reply indicates the rejection of a request.

(Responses in CoAP are replies that may carry a resource representation.)

A.4.3.3. Notifications

Matching the methods of an observer, the notification message types can be summarized as follows:

o A YIELD notification supplies the subscribed client with the state of a resource in some representation.

o A BREAK notification indicates that the observed resource has finished sending notifications.

o A THROW notification informs the subscribed client of an error condition.

Each of these notification messages can be sent as a message that does not require acknowledgment, as a confirmable message that does require acknowledgment (which makes it a request), or (in case of an initial notification) piggy-backed with the ACK message that is sent in reply to the subscription request.

A.5. Message Exchanges

The following message exchanges can be derived from the requirements.
A.5.1. Subscribing to a resource

The workflow for subscribing an observer to an observable resource or refreshing a subscription is as follows:

1. The subscribing client sends a SUBSCRIBE request to the server that is offering the observable resource. The subscription request includes the identifier of a representation format in which the notifications have to be sent, specifies a subscription lifetime duration and optionally indicates states cached by the client (by Etag or Date) and other information that might be relayed in a GET request.

2. The server creates a new subscription if no subscription exists (Appendix A.7), and changes the lifetime duration of the subscription to the duration specified in the subscription request, or any shorter duration if it so desires. The server then acknowledges the request and indicates the actual lifetime with a ACK reply (or with a ACK+THROW, ACK+BREAK or ACK+YIELD reply as described in the next step).

3. The server performs one of the following actions:

   * If an error occurred, the server sends a THROW notification (either as THROW request or as ACK+THROW reply to the SUBSCRIBE request).

   * If the resource has finished sending notifications, the server sends a BREAK notification (either as BREAK request or as ACK+...
BREAK reply to the SUBSCRIBE request).

* Otherwise, the server supplies the observer with the current resource state in the requested representation format, or indicates that the cached state is the current state (either as YIELD request or as ACK+YIELD reply to the SUBSCRIBE request).

If the server sends a THROW, BREAK or YIELD request, the request is treated like any other notification (i.e. the client must acknowledge it if the message is marked as confirmable, etc.).

4. If the client does not receive the ACK, ACK+THROW, ACK+BREAK or ACK+YIELD reply within a certain time frame (because the request or the reply went missing), the client retransmits the SUBSCRIBE request using the same request identifier.

A.5.2. Notifying of state changes

<table>
<thead>
<tr>
<th>Client</th>
<th>Server</th>
</tr>
</thead>
<tbody>
<tr>
<td>YIELD</td>
<td></td>
</tr>
<tr>
<td>&lt;=------------</td>
<td>or</td>
</tr>
<tr>
<td>ACK</td>
<td>YIELD [c]</td>
</tr>
<tr>
<td></td>
<td>&lt;=-----------</td>
</tr>
</tbody>
</table>

Figure 4

The workflow for notifying an observer of a state change is as follows:

1. The server performs one of the following actions:

* If the resource experienced an error condition, the server sends a THROW request to the client.

* If the resource has finished sending notifications, the server sends a BREAK request to the client.

* If the state of the resource changed, the server sends a YIELD request with the current resource state in the requested representation format.

The server may or may not mark the request sent as confirmable ("[c]"). The server marks a request as confirmable because it wants to check if the observer is still alive, or because there
might not be another notification in the near future and the confirmation process is therefore needed to ensure eventual consistency.

2. If the request is marked as confirmable, the subscribed client performs one of the following actions:

* If the notification cannot be related to a subscription, the client sends a RST reply.

* Otherwise, the client sends an ACK reply.

3. If the request is marked as confirmable and the server does not receive the ACK reply or the RST reply within a given time frame, the server retransmits the request using the same transaction identifier.

(Note that it is entirely the decision of the server whether to request an acknowledgment by marking the request as confirmable – it might want to employ different strategies to determine this, e.g. based on frequency of change, management of state etc.)

A.5.3. Unsubscribing from a resource

![Unsubscribing Flowchart](image)

The workflow for unsubscribing an observer from an observed resource is as follows:

1. The subscribed client sends an UNSUBSCRIBE request to the server offering the observed resource.

2. The server changes the lifetime duration of the subscription to zero, which immediately expires the subscription. The server then acknowledges the request with an ACK reply.

3. If the client does not receive the ACK reply within a certain time frame, the client retransmits the UNSUBSCRIBE request using the same request identifier.
Note that an subscribed client can also unsubscribe by "forgetting" the subscription and subsequently replying with a RST to the next notification. (In order to allow sending that RST even for messages that are not marked as confirmable, an otherwise redundant transaction identifier is sent in all messages.)

(An UNSUBSCRIBE message might be implemented as a SUBSCRIBE message with a lifetime duration of 0, but this has some unintended consequences. See Appendix A.7.)

A.5.4. Retrieving resource state

Note that a GET request matches closely the first steps of subscribing, except that the GET request does not create or update an subscription. (A GET request is likely to mark the yield as confirmable.)

The second form in Figure 6 solves the long-poll case.

```plaintext
Client             Server                  Client             Server
<p>| | | |
|                 |                        |                 |</p>
<table>
<thead>
<tr>
<th>GET</th>
<th></th>
<th>GET</th>
</tr>
</thead>
<tbody>
<tr>
<td>&lt;----------------</td>
<td></td>
<td>&lt;----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK+YIELD</td>
<td></td>
<td>ACK</td>
</tr>
<tr>
<td>&lt;----------------</td>
<td></td>
<td>&lt;----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td>or</td>
</tr>
<tr>
<td></td>
<td></td>
<td>... Time Passes ...</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>YIELD [c]</td>
<td></td>
<td>ACK</td>
</tr>
<tr>
<td>&lt;----------------</td>
<td></td>
<td>&lt;----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>ACK</td>
<td></td>
<td></td>
</tr>
<tr>
<td>&lt;----------------</td>
<td></td>
<td>&lt;----------------</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```

Figure 6
A.5.5. Changing resource state

![Diagram of resource state change process]

Figure 7

(The motivation for the BREAK component of the response is: PUT, like GET, can experience some error condition, so we need a place in the state machine to put the THROW response; therefore we also need a BREAK response. The second case in Figure 7 is motivated by: PUT, like GET, can take some time to be processed, so we must be able to send an ACK immediately and the THROW or BREAK response later.)

Obviously, changing the state of this resource leads to notification of any observers of new state. PUT is idempotent, but if we add notifications it may be a bit surprising that a duplicated or retransmitted PUT might send notifications twice. To prevent that, the resource might check whether it is being changed to the same state it had before and not send notifications in that case.

A.5.6. Deleting a resource

A DELETE request is the same as a PUT request, except that the resource is not created or its state changed, but deleted.
If a resource is observed, deleting the resource leads to an error condition of which the observers are notified with a THROW notification or THROW request.

A.6. Caching

An observer may cache part or all of a resource’s state changes. This allows clients to make conditional requests: If a subscribing client has cached a particular resource state, the server offering the resource does not need to send a full notification.

- A client informs the server of its cached resource states by including Etags of cached resource states in the subscription request. This can be multiple Etags, as this enables the server to just send a "304 not modified" for a different Etag instead of a different representation if the resource switches around between multiple states. (Note that this list might be updated by a subscription refresh.)

- A server notifies a client of a change to a cached state by omitting the representation of the resource state in the notification and including a flag ("304 not modified") that indicates that the state changed to a cached state.

Another approach to caching multiple values for a resource is to express each possible value of the resource’s representation by a reference to another (unchanging) resource. This lets a resource change states between a set of such references that then provide the actual state information. A subscribed client fetches the individual resources on demand and caches the results for future use.

A.7. Identifying notifications and subscriptions

There are two ways that a notification could be related to the resource that it is about:
1. The notification could name the resource (its URI).

2. The notification could name a target that relates to a subscription, which in turn relates back to the resource.

One or both ways could be implemented in CoAP, there are not

For way 1, each notification would contain the URI of the resource. This is particularly useful for multicast messages, but could be relatively wasteful. Also, it is not entirely clear that all servers will be aware of their own authority. Apart from cached states (Appendix A.6) and other information that could be part of a GET, a subscription would simply be the triple

[URI, observer transport address, lifetime]

Resubscribing (or a duplicate subscription request) for the same [URI, observer transport address] pair simply updates the lifetime; thus, the subscription operation is idempotent. Similarly, resubscribing with a lifetime of 0 will serve to delete the subscription (however, a SUBSCRIBE message will be replied to with the current state of the resource; a SUBSCRIBE with lifetime 0 is thus equivalent to a GET with the side effect of deleting the subscription).

The notification then needs to contain the URI (and might occasionally contain a remaining lifetime):

[resource representation, resource metadata, URI, lifetime?]

For way 2, the subscription request would also contain a target identifier that is to be used in every notification, identifying the specific observer object that is to receive the notification. Again apart from cached states (Appendix A.6) and other information that could be part of a GET, the subscription would be the quadruple:

[URI, observer transport address, target, lifetime]

(Note that the combination of the observer transport address and the target could be expressed as another URI, with the transport address as the authority and the target as the path. Alternatively, the target could be a short byte string. By varying the transport address, i.e. the port number, the target could also made to be always empty.)

Resubscribing (or a duplicate subscription request) for the same [URI, observer transport address, target] triple simply updates the lifetime, thus, the subscription operation again is idempotent. (The
same comment as with way 1 applies to SUBSCRIBE with lifetime 0.)

As the notification already implicitly contains the transport address, it would just list the target (and might occasionally contain a remaining lifetime):

[resource representation, resource metadata, target, lifetime?]

In both cases, it would be beneficial if rebooting nodes could obtain the same transport address they had before, because a resubscription after the reboot does not create additional state.

Multicast works nicely in way 1 (except that the URI may be large). In way 2, we can give each observer in a multicast group the same target name, so multicast works.

For normal subscriptions, the observer transport address can be implied from the source address of the subscription request. For multicast, there needs to be a way to explicitly indicate that transport address. (This may be beneficial for unicast, too.)

A.8. Open issues

Add discussion of messages that get reordered.

Add detailed message and option formats, once the semantics are agreed.

Describe how to map this to HTTP long-polls, WebSockets, and other asynchronous forms of HTTP.
Authors’ Addresses

Klaus Hartke
Universitaet Bremen T2I
Postfach 330440
Bremen  D-28359
Germany

Phone: +49-421-218-63905
Fax:   +49-421-218-7000
Email: hartke@tzi.org

Carsten Bormann
Universitaet Bremen T2I
Postfach 330440
Bremen  D-28359
Germany

Phone: +49-421-218-63921
Fax:   +49-421-218-7000
Email: cabo@tzi.org