CoRE Working Group
Internet-Draft
Intended status: Standards Track
Expires: August 29, 2013

K. Hartke
Universitaet Bremen TZI

February 25, 2013

Observing Resources in CoAP
draft-ietf-core-observe-08

Abstract

CoAP is a RESTful application protocol for constrained nodes and networks. The state of a resource on a CoAP server can change over time. This document specifies a simple protocol extension for CoAP that enables CoAP clients to "observe" resources, i.e., to retrieve a representation of a resource and keep this representation updated by the server over a period of time. The protocol follows a best-effort approach for sending new representations to clients, and provides eventual consistency between the state observed by each client and the actual resource state at the server.

Editor’s Note

This is an interim revision which will receive further modifications during the resolution of open tickets, in particular #204 and #281.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

1.1. Background

CoAP [I-D.ietf-core-coap] is an application protocol for constrained nodes and networks. It is intended to provide RESTful services [REST] not unlike HTTP [RFC2616] while reducing the complexity of implementation as well as the size of packets exchanged in order to make these services useful in a highly constrained network of themselves highly constrained nodes.

The model of REST is that of a client exchanging representations of resources with a server. A representation captures the current or intended state of a resource. The server is the definitive source for representations of the resources in its namespace. A client interested in the state of a resource initiates a request to the server; the server then returns a response with a representation of the resource that is current at the time of the request.

This model does not work well when a client is interested in having a current representation of a resource over a period of time. Existing approaches from HTTP, such as repeated polling or HTTP long polling [RFC6202], generate significant complexity and/or overhead and thus are less applicable in a constrained environment.

The protocol specified in this document extends the CoAP core protocol with a mechanism for a CoAP client to "observe" a resource on a CoAP server: the client can retrieve a representation of the resource and keep this representation updated by the server over a period of time.

The protocol keeps the architectural properties of REST. It enables high scalability and efficiency through the support of caches and proxies. There is no intention, though, to solve the full set of problems that the existing HTTP solutions solve, or to replace publish/subscribe networks that solve a much more general problem [RFC5989].

1.2. Protocol Overview

The protocol is based on the well-known observer design pattern [GOF]. In this design pattern, components called "observers" register at a specific, known provider called the "subject" that they are interested in being notified whenever the subject undergoes a change in state. The subject is responsible for administering its list of registered observers. If multiple subjects are of interest to an observer, it must register separately for all of them.

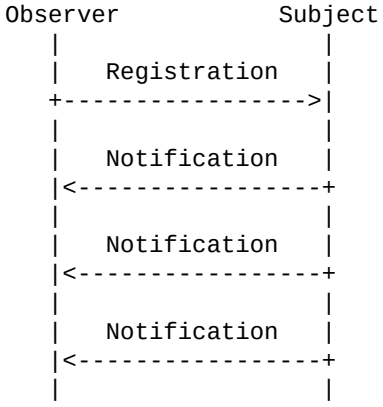

Figure 1: The Observer Design Pattern

The observer design pattern is realized in CoAP as follows:

Subject: In the context of CoAP, the subject is a resource in the namespace of a CoAP server. The state of the resource can change over time, ranging from infrequent updates to continuous state transformations.

Observer: An observer is a CoAP client that is interested in having a current representation of the resource at any given time.

Registration: A client registers its interest in a resource by initiating an extended GET request to the server. In addition to returning a representation of the target resource, this request causes the server to add the client to the list of observers of the resource.

Notification: Whenever the state of a resource changes, the server notifies each client in the list of observers of the resource. Each notification is an additional CoAP response sent by the server in reply to the GET request and includes a complete, updated representation of the new resource state.

Figure 2 below shows an example of a CoAP client registering its interest in a resource and receiving three notifications: the first upon registration with the current state, and then two upon two changes to the resource state. Both the registration request and the notifications are identified as such by the presence of the Observe Option defined in this document. In notifications, the Observe Option provides a sequence number for reordering detection. All notifications carry the token specified by the client in the request, so the client can easily correlate them to the request.

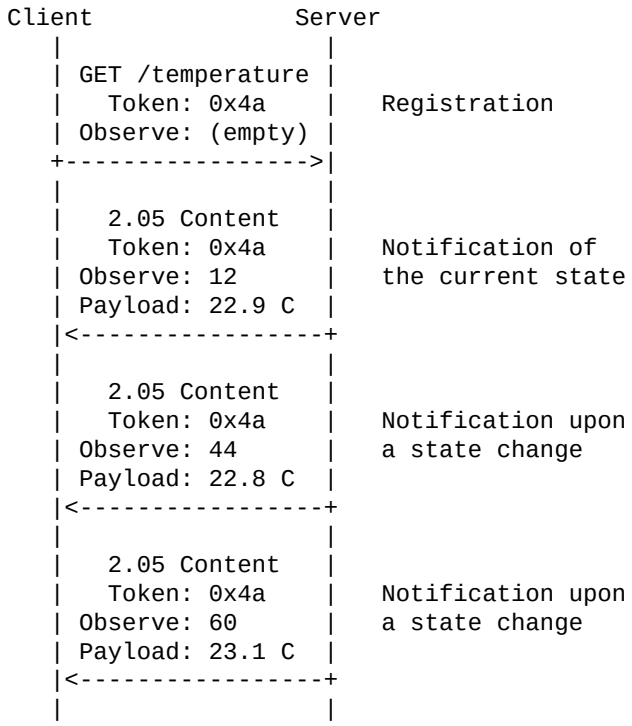

Figure 2: Observing a Resource in CoAP

The server is the authority for determining under what conditions resources change their state and how often observers are notified. The protocol does not offer explicit means for setting up triggers, thresholds or other conditions; it is up to the server to expose observable resources that change their state in a way that is useful in the application context. Resources can be parameterized to achieve similar effects, though; see Appendix B for examples.

A client remains on the list of observers as long as the server can determine the client's continued interest in the resource. The interest is determined from the client's acknowledgement of notifications sent in confirmable messages by the server. If the client actively rejects a notification or if the transmission of a notification ultimately fails, then the client is assumed to be no longer interested and is removed from the list of observers.

While a client is in the list of observers of a resource, it is the goal of the protocol to keep the resource state observed by the client as closely in sync with the actual state at the server as possible.

Becoming out of sync at times cannot be avoided: First, there is always some latency between the change of the resource state and the receipt of the notification. Second, messages with notifications can get lost, which will cause the client assume an old state until it receives the next notification. And third, the server may erroneously come to the conclusion that the client is no longer interested in the resource, which will cause it to stop sending notifications and the client to assume an old state until it registers its interest again.

The protocol addresses this issue as follows:

- It follows a best-effort approach for sending the current representation to the client after a state change: Clients should see the new state after a state change as soon as possible, and they should see as many states as possible. However, a client cannot rely on observing every single state that a resource goes through.

- It labels notifications with a maximum duration up to which it is acceptable for the observed state and the actual state to be out of sync. When the age of the notification received reaches this maximum, the client cannot use the enclosed representation until it has received a new notification.

- It is designed on the principle of eventual consistency: The protocol 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.

1.3. Requirements Notation

The key words "MUST", "MUST NOT", "REQUIRED", "SHALL", "SHALL NOT", "SHOULD", "SHOULD NOT", "RECOMMENDED", "MAY", and "OPTIONAL" in this document are to be interpreted as described in RFC 2119 [RFC2119].

2. The Observe Option

No.	C	U	N	R	Name	Format	Length	Default
6		x	-		Observe	empty/uint	0 B/0-3 B	(none)

C=Critical, U=Unsafe, N=No-Cache-Key, R=Repeatable

Table 1: The Observe Option

The Observe Option, when present in a request, extends the GET method so it does not only retrieve a current representation of the target resource, but also requests the server to add the client to the list of observers of the resource. The exact semantics are defined in the following sections. The value of the option in a request MUST be empty on transmission and MUST be ignored on reception.

The Observe Option is not critical for processing the request. If the server is unwilling or unable to add the client to the list of observers of the resource identified by the request URI, then the request falls back to a normal GET request.

In a response, the Observe Option identifies the message as a notification. This implies that the server has added the client to the list of observers and that it will notify the client of changes to the resource state. The value of the option is a 24-bit sequence number for reordering detection; see Section 3.4 and Section 4.4 for the client- and server-side respectively. The sequence number is encoded in network byte order using a variable number of bytes, as specified in Section 3.2 of RFC XXXX [I-D.ietf-core-coap].

The Observe Option is not part of the cache-key: a cacheable response obtained with an Observe Option in the request can be used to satisfy a request without an Observe Option, and vice versa. When a stored response with an Observe Option is used to satisfy a normal GET request, the option MUST be removed before the response is returned to the client.

3. Client-side Requirements

3.1. Request

A client can register its interest in a resource by issuing a GET request that includes an empty Observe Option. If the server returns a 2.xx response that includes an Observe Option as well, the server has added the client successfully to the list of observers of the target resource and the client will be notified of changes to the resource state.

3.2. Notifications

Notifications are additional responses sent by the server in reply to the GET request. Each notification includes the token specified by the client in the GET request, an Observe Option with a sequence number for reordering detection (see Section 3.4) and a payload in the same Content-Format as the initial response.

Notifications have a 2.05 (Content) response code, or a 2.03 (Valid) response code if the client has included one or more ETag Options in the request (see Section 3.3). In the event that the resource changes in a way that would cause a normal GET request at that time to return a non-2.xx response (for example, when the resource is deleted), the server sends a notification with a matching response code and removes the client from the list of observers.

3.3. Caching

As notifications are just additional responses to a GET request, notifications partake in caching as defined in Section 5.6 of RFC XXXX [I-D.ietf-core-coap]. Both the freshness model and the validation model are supported.

3.3.1. Freshness

A client MAY store a notification like a response in its cache and use a stored notification that is fresh without contacting the origin server. Like a response, a notification is considered fresh while its age is not greater than the value indicated by the Max-Age Option and if no newer notification/response has been received.

The server will do its best to keep the resource state observed by the client as closely in sync with the actual state as possible. However, a client cannot rely on observing every single state that a resource might go through. For example, if the network is congested or the state changes more frequently than the network can handle, the server can skip notifications for intermediate states.

The server uses the Max-Age Option to indicate an age up to which it is acceptable that the observed state and the actual state are inconsistent. If the age of the latest notification becomes greater than its indicated Max-Age, then the client MUST NOT use the enclosed representation until it is validated or a new notification is received.

To make sure it has a current representation and/or to re-register its interest in a resource, a client MAY issue a new GET request with an Observe Option at any time. The client SHOULD specify a new token in the GET request, as the token serves as an epoch identifier for the sequence numbers in the Observe Option (see Section 3.4).

It is RECOMMENDED that the client does not issue the request while it still has a fresh notification/response for a resource in its cache. Additionally, the client SHOULD wait for a random amount of time between 5 and 15 seconds before issuing the new request to avoid synchronicity with other clients.

3.3.2. Validation

When a client has one or more notifications stored in its cache for a resource, it can use the ETag Option in the GET request to give the server an opportunity to select a stored notification to be used.

The client MAY include an ETag Option for each stored response that is applicable in the GET request. Whenever the observed resource changes to a representation identified by one of the ETag Options, the server can select a stored response by sending a 2.03 (Valid) notification with an appropriate ETag Option instead of a 2.05 (Content) notification. The client needs to keep all candidate responses in its cache until it is no longer interested in the target resource or it issues a new GET request with a new set of entity-tags.

3.4. Reordering

Messages with notifications can arrive in a different order than they were sent. Since the goal is to keep the observed state as closely in sync with the actual state as possible, a client SHOULD NOT update the observed state with a notification that arrives later than a newer notification.

For reordering detection, the server sets the value of the Observe Option in each notification to a 24-bit sequence number. An incoming notification is newer than the newest notification received so far when one of the following conditions is met:

```
(V1 < V2 and V2 - V1 < 2^23) or
(V1 > V2 and V1 - V2 > 2^23) or
(T2 > T1 + 128 seconds)
```

where V1 is the value of the Observe Option of the newest notification received so far, V2 the value of the Observe Option of the incoming notification, T1 a client-local timestamp of the newest notification received so far, and T2 a client-local timestamp of the incoming notification.

Design Note: The first two conditions verify that V1 is less than V2 in 24-bit sequence number arithmetic [RFC1982]. The third condition ensures that the time elapsed between the two incoming messages is not so large that the difference between V1 and V2 has become larger than the largest integer that it is meaningful to add to a 24-bit sequence number; in other words, after 128 seconds have elapsed without any notification, a client does not need to check the sequence numbers in order to assume an incoming notification is new.

3.5. Transmission

A notification can be confirmable or non-confirmable, i.e., be sent in a confirmable or a non-confirmable message. The message type used is independent from the type used for the request or any previous notification.

If a client does not recognize the token in a confirmable notification, it MUST NOT acknowledge the message and SHOULD reject it with a Reset message; otherwise, the client MUST acknowledge the message as usual. In the case of a non-confirmable notification, rejecting the message with a Reset message is OPTIONAL.

An acknowledgement message signals to the server that the client is alive and interested in receiving further notifications; if the server does not receive an acknowledgement in reply to a confirmable notification, it will assume that the client is no longer interested and will eventually remove the client from the list of observers.

3.6. Cancellation

When a client rejects a confirmable notification with a Reset message or when it issues a GET request without an Observe Option for a currently observed resource, the server will remove the client from the list of observers of this resource. The client MAY use either method to indicate that it is no longer interested in receiving further notifications for the resource until it eventually registers again.

When a client rejects non-confirmable notification, the server may also (but is not required to) remove the client from the list of observers of this resource. The client MAY try this method as well, and MAY rate-limit the Reset messages it sends if the server appears to persistently ignore them.

Implementation Note: A client that does not mediate all its requests through its cache might inadvertently cancel an observation by making an unrelated GET to the same resource. To avoid this, without incurring a need for synchronization, such clients can use a different source endpoint for these unrelated GET requests.

4. Server-side Requirements

4.1. Request

A GET request that includes an Observe Option requests the server not only to return a current representation of the resource identified by

the request URI, but also to add the client to the list of observers of the target resource. If no error occurs, the server MUST return a 2.05 (Content) response with the representation of the current resource state and MUST notify the client of subsequent changes to the state as long as the client is on the list of observers.

A server that is unable or unwilling to add the client to the list of observers of the target resource MAY silently ignore the Observe Option and process the GET request as usual. The resulting response MUST NOT include an Observe Option, the absence of which signals to the client that it will not be notified of changes to the resource and, e.g., needs to poll the resource for its state instead.

If the client is already on the list of observers, the server MUST NOT add it a second time but MUST replace or update the existing entry. If the server receives a GET request for the a resource that does not include an Observe Option, the server MUST remove any existing entry from the list of observers.

Two requests relate to the same list entry if and only if both the request URI and the source endpoint of the requests are the same. Message IDs, tokens and other options are not taken into account.

Any request with an unrecognized critical option or a method other than GET MUST NOT have a direct effect on a list of observers of a resource. However, a non-GET request can have the indirect consequence of causing the server to send a non-2.xx notification which does affect the list of observers (for example, when a DELETE request is successful and the observed resource no longer exists).

4.2. Notifications

A client is notified of changes to the resource state by additional responses sent by the server in reply to the GET request. Each such notification response (including the initial response) MUST include an Observe Option and MUST echo the token specified by the client in the GET request. If there are multiple clients on the list of observers, the order in which they are notified is not defined; the server is free to use any method to determine the order.

A notification SHOULD have a 2.05 (Content) or 2.03 (Valid) response code. However, in the event that the state of a resource changes in a way that would cause a normal GET request at that time to return a non-2.xx response (for example, when the resource is deleted), the server SHOULD notify the client by sending a notification with a matching response code and MUST remove the client from the list of observers of the resource.

The Content-Format used in a notification MUST be the same as the one used in the initial response to the GET request. If the server is unable to continue sending notifications using this Content-Format, it SHOULD send a notification with a 5.00 (Internal Server Error) response code and MUST remove the client from the list of observers of the resource.

4.3. Caching

As notifications are just additional responses sent by the server, they are subject to caching as defined in Section 5.6 of RFC XXXX [I-D.ietf-core-coap].

4.3.1. Freshness

After returning the initial response, the server MUST try to keep the returned representation current, i.e., keep the resource state observed by the client as closely in sync with the actual resource state as possible.

Since becoming out of sync at times cannot be avoided, the server MUST indicate for each representation an age up to which it is acceptable that the observed state and the actual state are inconsistent. This age is application-dependent and MUST be specified in notifications using the Max-Age Option.

When the resource does not change and the client has a current representation, the server does not need to send a notification. However, if the client does not receive a notification, it cannot tell if the observed state and the actual state are still in sync. So, when the the age of the latest notification becomes greater than its indicated Max-Age, then the client will assume that the states are inconsistent until the representation is validated or a new notification is received. The server MAY wish to prevent that by sending a notification with the unchanged representation before Max-Age expires.

4.3.2. Validation

A client can include a set of entity-tags in its request using the ETag Option. When a observed resource changes its state and the origin server is about to send a 2.05 (Content) notification, then, whenever that notification has an entity-tag in the set of entity-tags specified by the client, the server MAY send a 2.03 (Valid) response with an appropriate ETag Option instead. The server MUST NOT assume that the client has any response stored other than those identified by the entity-tags in the most recent GET request received for the resource.

4.4. Reordering

Because messages can get reordered, the client needs a way to determine if a notification arrived later than a newer notification. For this purpose, the server MUST set the value of the Observe Option of each notification it sends to the 24 least-significant bits of a strictly increasing sequence number. The sequence number MAY start at any value and MUST NOT increase so fast that it increases by more than 2^24 within less than 256 seconds.

The sequence number selected for a notification MUST be greater than that of any preceding notification sent to the same client for the same resource with the same token. The value of the Observe Option MUST be current at the time of transmission; if a notification is retransmitted, the server MUST update value of the Observe Option before sending the message.

Implementation Note: A simple implementation that satisfies the requirements is to obtain a timestamp from a local clock. The sequence number then is the timestamp in ticks, where 1 tick = (256 seconds)/(2^24) = 15.26 microseconds. It is not necessary that the clock reflects the current time/date or that it ticks in a precisely periodical way.

Another valid implementation is to store a 24-bit unsigned integer variable per resource and increment this variable each time the resource undergoes a change of state (provided that the resource changes its state less than 2^24 times in the next 256 seconds after any state change). This alleviates the need to update the value of the Observe Option in a message when the resource state did not change.

Design Note: The choice of a 24-bit option value and a time span of 256 seconds allows for a notification rate of up to 65536 notifications per second. 64K ought to be enough for anybody.

4.5. Transmission

A notification can be sent in a confirmable or a non-confirmable message. The message type used is typically application-dependent and MAY be determined by the server for each notification individually. For example, for resources that change in a somewhat predictable or regular fashion, notifications can be sent in non-confirmable messages; for resources that change infrequently, notifications can be sent in confirmable messages. The server can combine these two approaches depending on the frequency of state changes and the importance of individual notifications.

The acknowledgement of a confirmable notification signals to the server that the client is interested in receiving further notifications. If a client rejects a confirmable notification with a Reset message, the client is no longer interested and the server MUST remove the client from the list of observers. If the client rejects a non-confirmable notification, the server MAY remove the client from the list of observers as well. (It is expected that the server does remove the client if it has the information available that is needed to match the Reset message to the non-confirmable notification, but the server is not required to keep this information.)

At a minimum, the server MUST send a notification in a confirmable message instead of a non-confirmable message at least every 24 hours.

A server MAY choose to skip a notification if it knows that it will send another notification soon (e.g., when the state is changing frequently). Similarly, it MAY choose to send a notification more than once. For example, when state changes occur in bursts, the server can skip some notifications, send the notifications in non-confirmable messages, and make sure that the client observes the latest state change after the burst by repeating the last notification in a confirmable message.

The server MUST limit the number of confirmable notifications for which an acknowledgement has not been received yet to NSTART (see Section 4.7 of RFC XXXX [I-D.ietf-core-coap]), and it SHOULD NOT send more than one non-confirmable notification every 3 seconds on average.

When the state of an observed resource changes while the server is still waiting for a confirmable notification to be acknowledged or the 3 seconds for a non-confirmable notification to elapse, then the server MUST proceed as follows:

1. Wait for the current transmission attempt to complete.

2. If the result is a Reset message or the transmission was the last attempt to deliver a notification, remove the client from the list of observers of the observed resource.

3. If the client is still in the list of observers, transmit a notification with a representation of the current resource state. Should the resource have changed its state more than once in the meantime, skip the notifications for the intermediate states.

4. If the previously completed transmission timed out, increment the retransmission counter and double the timeout; otherwise, reinitialize the retransmission counter and the timeout.

If CoAP is used over a connection-oriented or session-based transport such as DTLS, the server MUST remove the client from the list of observers when the connection or session is closed.

5. Intermediaries

A client may be interested in a resource in the namespace of an origin server that is reached through a chain of one or more CoAP-to-CoAP intermediaries. In this case, the client registers its interest with the first intermediary towards the origin server, acting as if it was communicating with the origin server itself as specified in Section 3. It is the task of this intermediary to provide the client with a current representation of the target resource and send notifications upon changes to the target resource state, much like an origin server as specified in Section 4.

To perform this task, the intermediary SHOULD make use of the protocol specified in this document, taking the role of the client and registering its own interest in the target resource with the next hop towards the origin server. If the next hop does not return a response with an Observe Option, the intermediary MAY resort to polling the next hop or MAY itself return a response without an Observe Option.

The communication between each pair of hops is independent; each hop in the server role MUST determine individually how many notifications to send, of which message type, and so on. Each hop MUST generate its own values for the Observe Option, and MUST set the value of the Max-Age Option according to the age of the local current representation.

Because a client (or an intermediary in the client role) can only be once on the list of observers of a resource on a server (or an intermediary in the server role) -- it is useless to observe the same resource multiple times -- an intermediary MUST observe a resource only once, even if there are multiple clients for which it observes the resource.

An intermediary is not required to act on behalf of a client to observe a resource; an intermediary MAY observe a resource, for example, just to keep its own cache up to date.

See Appendix A.1 for examples.

6. Blockwise Transfers

Resources observed by clients may be larger than can be comfortably processed or transferred in one CoAP message. CoAP provides a blockwise transfer mechanism to address this problem [I-D.ietf-core-block]. The following rules apply to the combination of blockwise transfers with notifications.

As with basic GET transfers, the client can indicate its desired block size in a Block2 Option in the GET request. If the server supports blockwise transfers, it SHOULD take note of the block size for all notifications/responses resulting from the GET request (until the client is removed from the list of observers or the server receives a new GET request for the resource from the client).

When sending a 2.05 (Content) notification, the server always sends all blocks of the representation, suitably sequenced by its congestion control mechanism, even if only some of the blocks have changed with respect to a previous notification. The server performs the blockwise transfer by making use of the Block2 Option in each block. When reassembling representations that are transmitted in multiple blocks, the client MUST NOT combine blocks carrying different Observe Option values.

Blockwise transfers of notifications MUST use confirmable messages and MUST NOT use non-confirmable messages.

See Appendix A.2 for an example.

7. Web Linking

A web link [RFC5988] to a resource accessible over CoAP (for example, in a link-format document [RFC6690]) MAY include the target attribute "obs".

The "obs" attribute, when present, is a hint indicating that the destination of a link is useful for observation and thus, for example, should have a suitable graphical representation in a user interface. Note that this is only a hint; it is not a promise that the Observe Option can actually be used to perform the observation. A client may need to resort to polling the resource if the Observe Option is not returned in the response to the GET request.

A value MUST NOT be given for the "obs" attribute; any present value MUST be ignored by parsers. The "obs" attribute MUST NOT appear more than once in a given link-value; occurrences after the first MUST be ignored by parsers.

8. Security Considerations

The security considerations of RFC XXXX [I-D.ietf-core-coap] apply.

The considerations about amplification attacks are somewhat amplified when observing resources. Without client authentication, a server MUST therefore strictly limit the number of notifications that it sends between receiving acknowledgements that confirm the actual interest of the client in the data; i.e., any notifications sent in non-confirmable messages MUST be interspersed with confirmable messages. (An attacker may still spoof the acknowledgements if the confirmable messages are sufficiently predictable.)

As with any protocol that creates state, attackers may attempt to exhaust the resources that the server has available for maintaining the list of observers for each resource. Servers may want to access-control this creation of state. As degraded behavior, the server can always fall back to processing the request as a normal GET request (without an Observe Option) if it is unwilling or unable to add a client to the list of observers of a resource, including if system resources are exhausted or nearing exhaustion.

Intermediaries must be careful to ensure that notifications cannot be employed to create a loop. A simple way to break any loops is to employ caches for forwarding notifications in intermediaries.

9. IANA Considerations

The following entry is added to the CoAP Option Numbers registry:

Number	Name	Reference
6	Observe	[RFCXXXX]

10. Acknowledgements

Carsten Bormann was an original author of this draft and is acknowledged for significant contribution to this document.

Thanks to Daniele Alessandrelli, Jari Arkko, Peter Bigot, Angelo P. Castellani, Gilbert Clark, Esko Dijk, Thomas Fossati, Brian Frank, Jeroen Hoebeke, Cullen Jennings, Matthias Kovatsch, Salvatore Loreto, Charles Palmer, Zach Shelby and Floris Van den Abeele for helpful comments and discussions that have shaped the document.

11. References

11.1. Normative References

[I-D.ietf-core-block]
Bormann, C. and Z. Shelby, "Blockwise transfers in CoAP", draft-ietf-core-block-10 (work in progress), October 2012.

[I-D.ietf-core-coap]
Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-13 (work in progress), December 2012.

[RFC1982] Elz, R. and R. Bush, "Serial Number Arithmetic", RFC 1982, August 1996.

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.

[RFC5988] Nottingham, M., "Web Linking", RFC 5988, October 2010.

11.2. Informative References

[GOF] Gamma, E., Helm, R., Johnson, R., and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley, Reading, MA, USA, November 1994.

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", Ph.D. Dissertation, University of California, Irvine, 2000, <http://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf>.

[RFC2616] Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.

[RFC5989] Roach, A., "A SIP Event Package for Subscribing to Changes to an HTTP Resource", RFC 5989, October 2010.

[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, April 2011.

[RFC6690] Shelby, Z., "Constrained RESTful Environments (CoRE) Link Format", RFC 6690, August 2012.

Appendix A. Examples

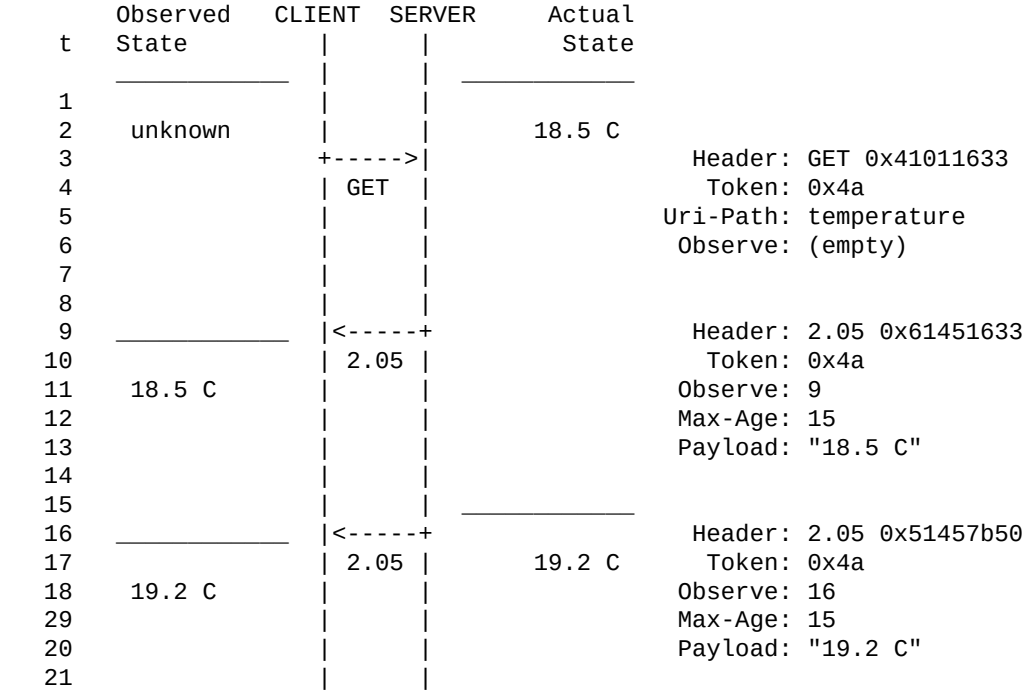

Figure 3: A client registers and receives one notification of the current state and one of a new state upon a state change

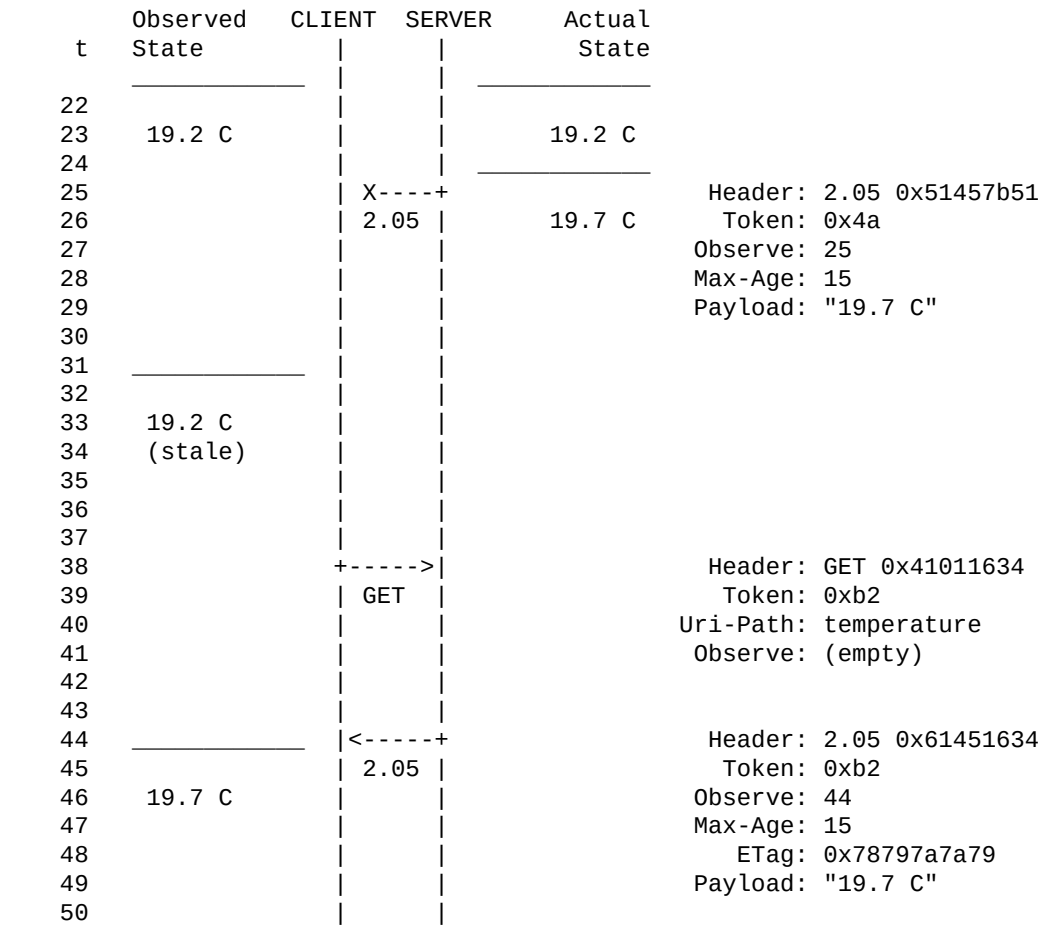

Figure 4: The client re-registers after Max-Age ends

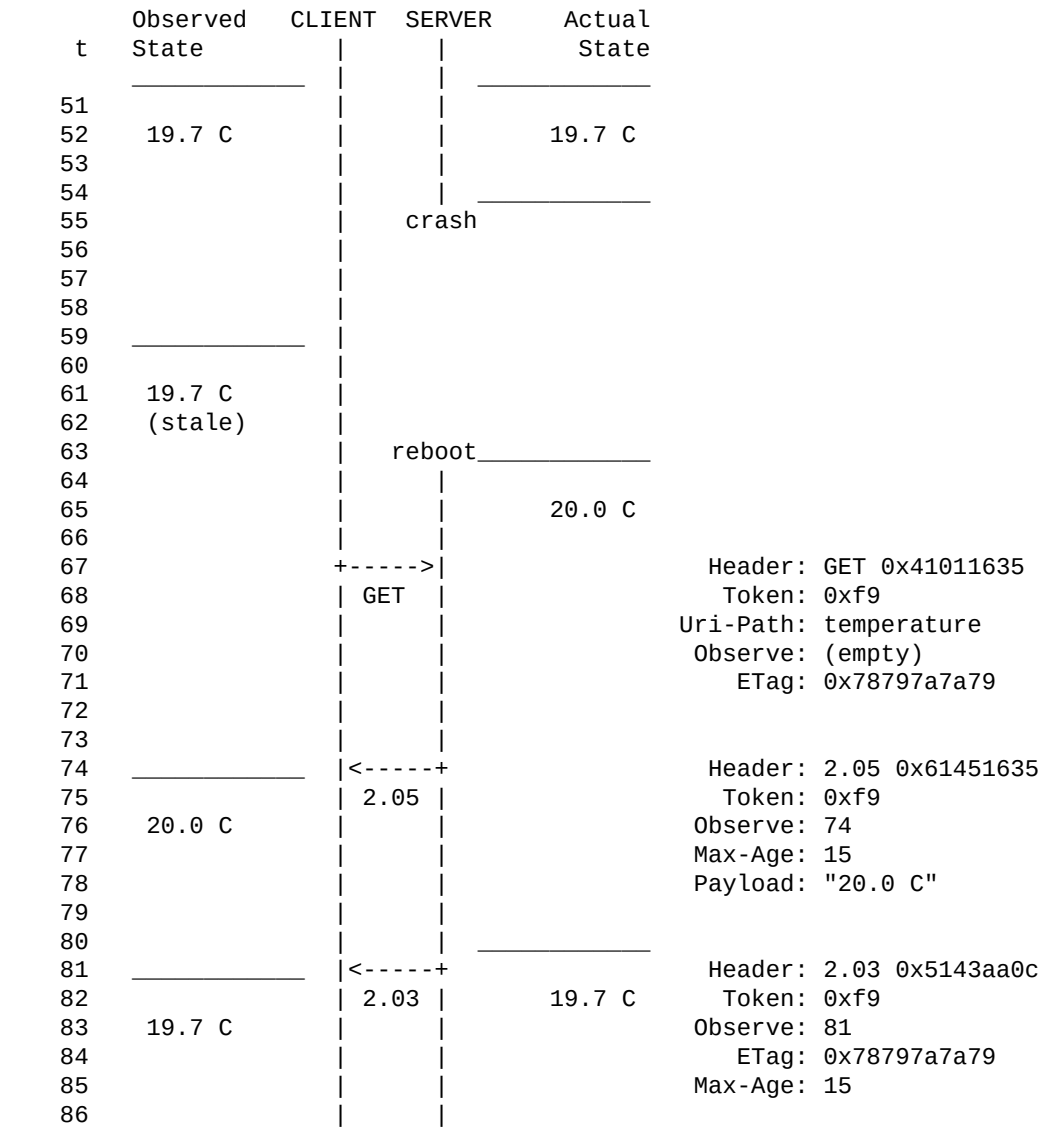

Figure 5: The client re-registers and gives the server the opportunity to select a stored response

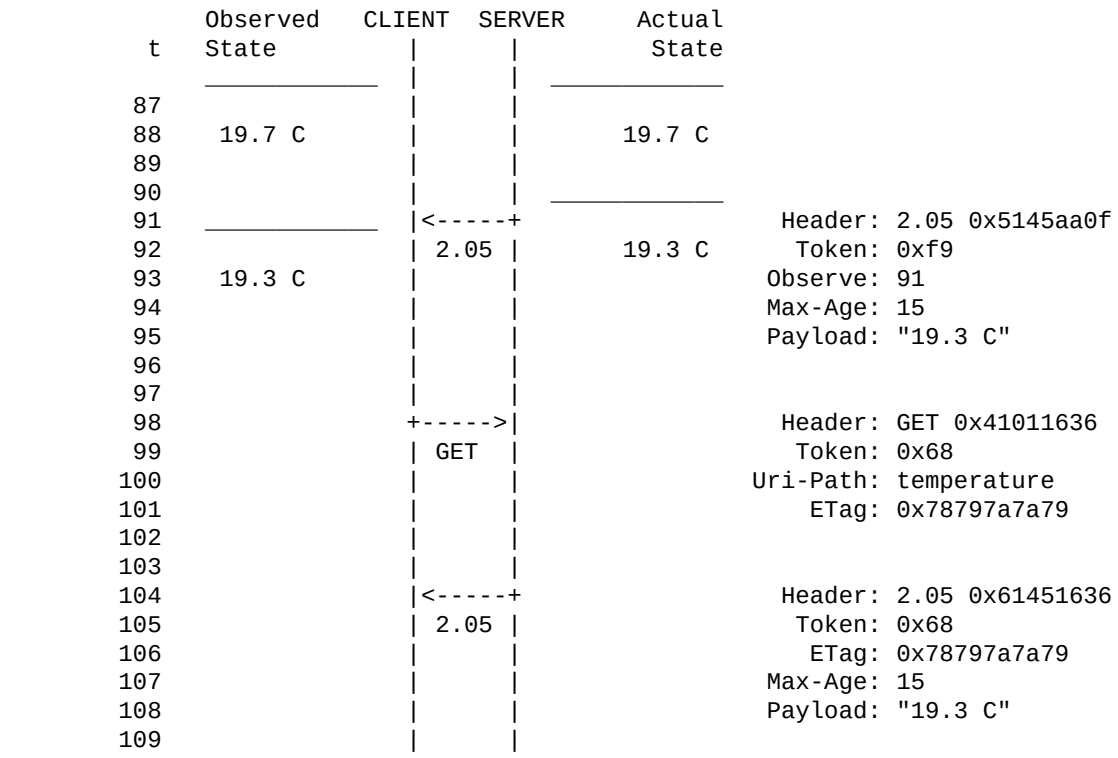

Figure 6: The client makes a normal GET request and thereby cancels the observation

A.1. Proxying

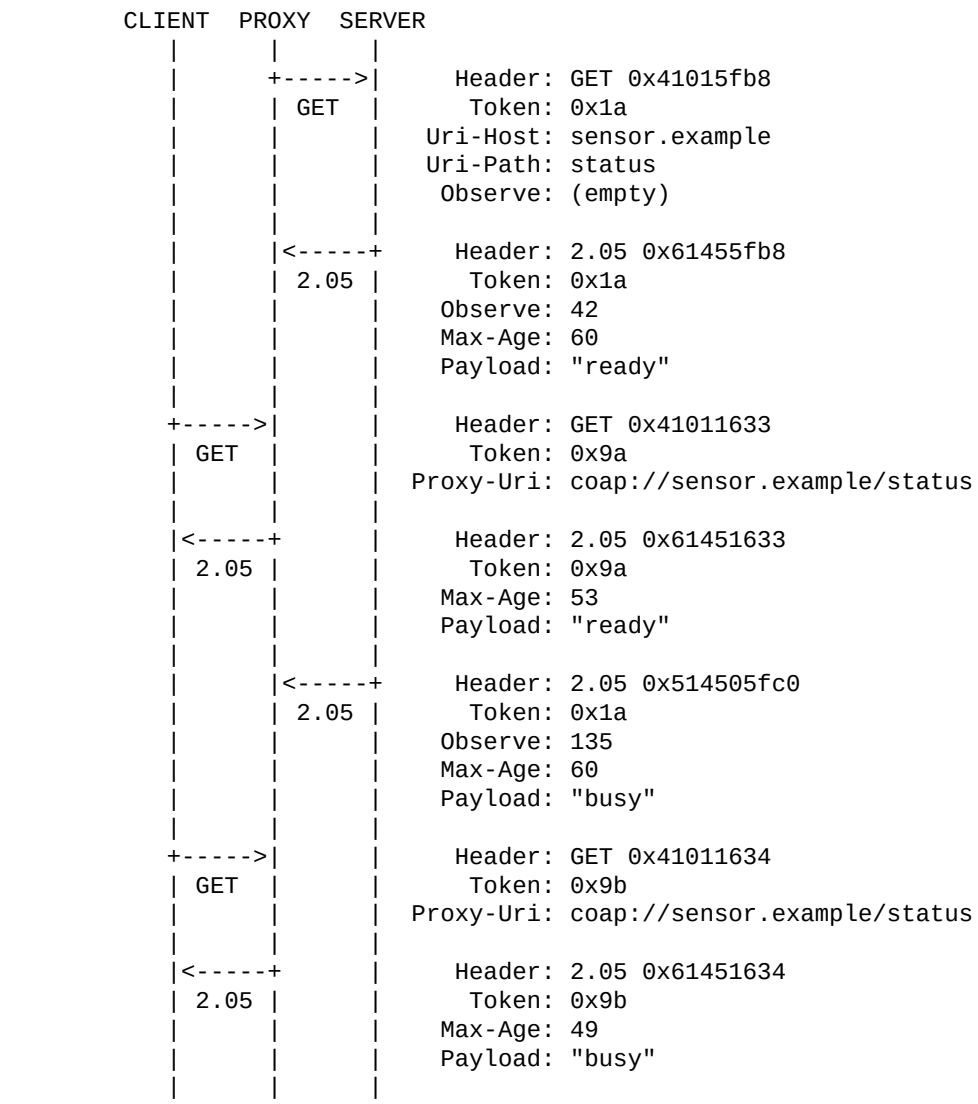

Figure 7: A proxy observes a resource to keep its cache up to date

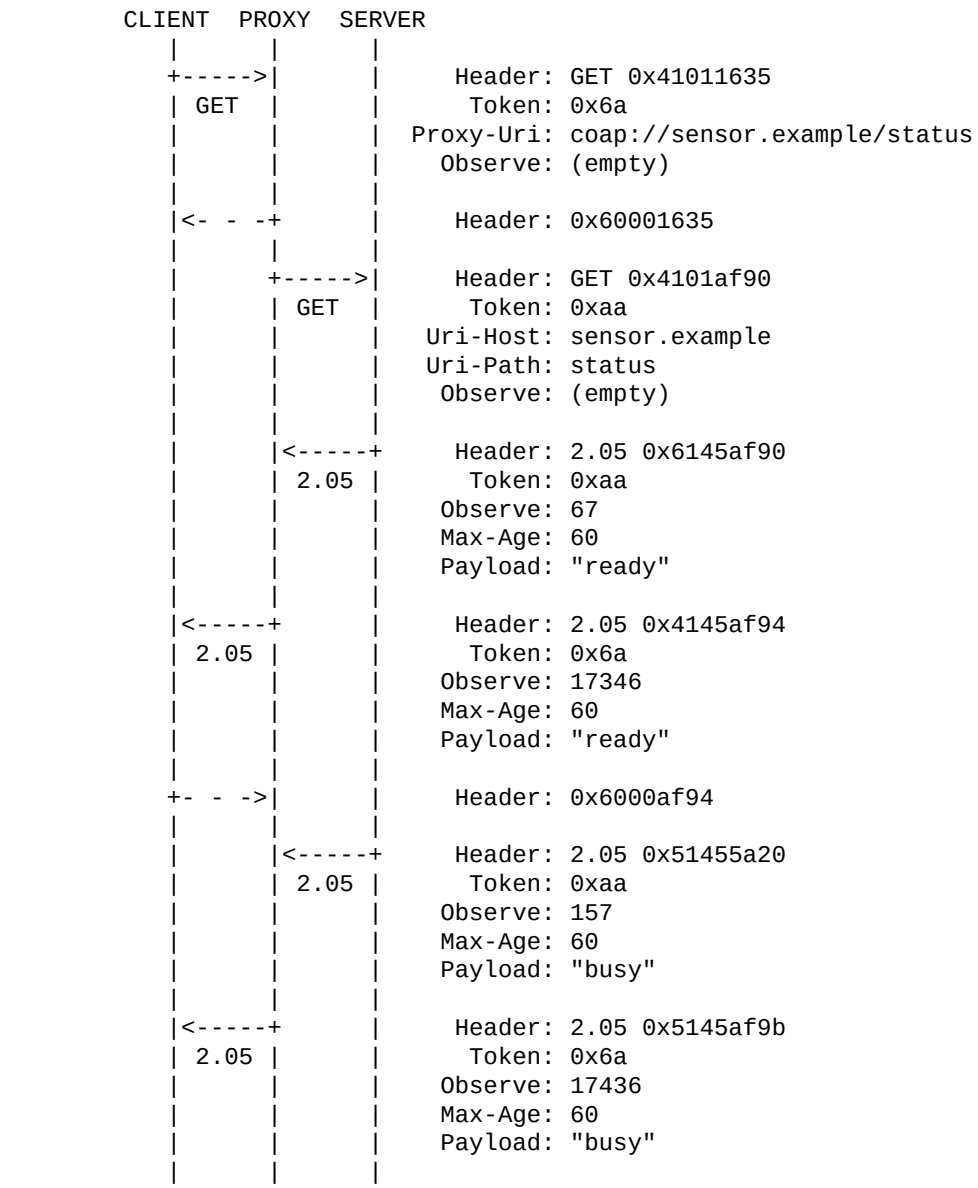

Figure 8: A client observes a resource through a proxy

A.2. Blockwise Transfer

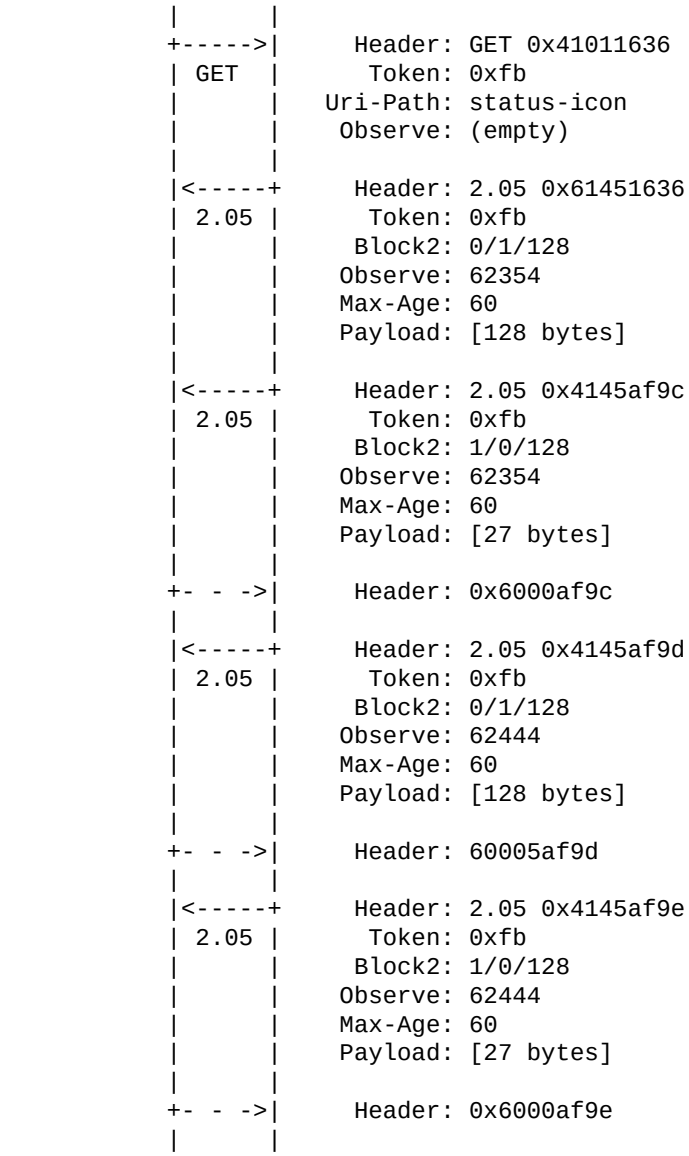

Figure 9: A server sends two notifications of two blocks each

Appendix B. Modeling Resources to Tailor Notifications

A server may want to provide notifications that respond to very specific conditions on some state. This is best done by modeling the resources that the server exposes according to these needs.

For example, for a CoAP server with an attached temperature sensor,

- the server could, in the simplest form, expose a resource <coap://server/temperature> that changes its state every second to the current temperature measured by the sensor;

- the server could, however, also expose a resource <coap://server/temperature/felt> that changes its state to "cold" when it's warm and the temperature drops below a preconfigured threshold, and to "warm" when it's cold and the temperature exceeds a second, higher threshold;

- the server could expose a parameterized resource <coap://server/temperature/critical?above=45> that changes its state every second to the current temperature if the sensor reading exceeds the specified parameter value, and that changes its state to "OK" when the temperature drops below; or

- the server could expose a parameterized resource <coap://server/temperature?query=select+avg(temperature)+from+ Sensor.window:time(30sec)> that accepts expressions of arbitrary complexity and changes its state accordingly.

In any case, the client is notified about the current state of the resource whenever the state of the appropriately modeled resource changes. By designing resources that change their state on certain conditions, it is possible to notify the client only when these conditions occur instead of continuously supplying it with information it doesn't need.

By parametrizing resources, this is not limited to conditions defined by the server, but can be extended to arbitrarily complex conditions defined by the client. Thus, the server designer can choose exactly the right level of complexity for the application envisioned and devices used, and is not constrained to a "one size fits all" mechanism built into the protocol.

Appendix C. Changelog

Changes from ietf-07 to ietf-08:

- Expanded text on transmitting notification while a previous transmission is pending (#242).

- Changed reordering detection to use a fixed time span of 128 seconds instead of EXCHANGE_LIFETIME (#276).

- Removed the use of the freshness model to determine if the client is still on the list of observers. This includes removing that

  * the client assumes that it has been removed from the list of observers when Max-Age ends;

  * the server sets the Max-Age Option of a notification to a value that indicates when the server will send the next notification;

  * the server uses a number of retransmit attempts such that removing a client from the list of observers before Max-Age ends is avoided (#235);

  * the server may remove the client from all lists of observers when the transmission of a confirmable notification ultimately times out.

- Changed that an unrecognized critical option in a request must actually have no effect on the state of any observation relationship to any resource, as the option could lead to a different target resource.

- Clarified that client implementations must be prepared to receive each notification equally as a confirmable or a non-confirmable message, regardless of the message type of the request and of any previous notification.

- Added a requirement for sending a confirmable notification at least every 24 hours before continuing with non-confirmable notifications (#221).

- Added congestion control considerations from [I-D.bormann-core-congestion-control-02].

- Recommended that the client waits for a randomized time after the freshness of the latest notification expired before re-registering. This prevents that multiple clients observing a resource perform a GET request at the same time when the need to re-register arises.

- Changed reordering detection from ’MAY’ to ’SHOULD’, as the goal of the protocol (to keep the observed state as closely in sync

with the actual state as possible) is not optional.

- Fixed the length of the Observe (3 bytes) in the table in Section 2.
- Replaced the 'x' in the No-Cache-Key column in the table in Section 2 with a '-', as the Observe Option doesn't have the No-Cache-Key flag set, even though it is not part of the cache key.
- Updated examples.

Changes from ietf-06 to ietf-07:

- Moved to 24-bit sequence numbers to allow for up to 15000 notifications per second per client and resource (#217).
- Re-numbered option number to use Unsafe/Safe and Cache-Key compliant numbers (#241).
- Clarified how to react to a Reset message that is sent in reply to a non-confirmable notification (#225).
- Clarified the semantics of the "obs" link target attribute (#236).

Changes from ietf-05 to ietf-06:

- Improved abstract and introduction to say that the protocol is about best effort and eventual consistency (#219).
- Clarified that the value of the Observe Option in a request must have zero length.
- Added requirement that the sequence number must be updated each time a server retransmits a notification.
- Clarified that a server must remove a client from the list of observers when it receives a GET request with an unrecognized critical option.
- Updated the text to use the endpoint concept from [I-D.ietf-core-coap] (#224).
- Improved the reordering text (#223).

Changes from ietf-04 to ietf-05:

- Recommended that a client does not re-register while a new notification from the server is still likely to arrive. This is

to avoid that the request of the client and the last notification after max-age cross over each other (#174).

- Relaxed requirements when sending a Reset message in reply to non-confirmable notifications.
- Added an implementation note about careless GET requests (#184).
- Updated examples.

Changes from ietf-03 to ietf-04:

- Removed the "Max-OFE" Option.
- Allowed a Reset message in reply to non-confirmable notifications.
- Added a section on cancellation.
- Updated examples.

Changes from ietf-02 to ietf-03:

- Separated client-side and server-side requirements.
- Fixed uncertainty if client is still on the list of observers by introducing a liveliness model based on Max-Age and a new option called "Max-OFE" (#174).
- Simplified the text on message reordering (#129).
- Clarified requirements for intermediaries.
- Clarified the combination of blockwise transfers with notifications (#172).
- Updated examples to show how the state observed by the client becomes eventually consistent with the actual state on the server.
- Added examples for parameterization of observable resource.

Changes from ietf-01 to ietf-02:

- Removed the requirement of periodic refreshing (#126).
- The new "Observe" Option replaces the "Lifetime" Option.
- Introduced a new mechanism to detect message reordering.

- Changed 2.00 (OK) notifications to 2.05 (Content) notifications.

Changes from ietf-00 to ietf-01:

- Changed terminology from "subscriptions" to "observation relationships" (#33).
- Changed the name of the option to "Lifetime".
- Clarified establishment of observation relationships.
- Clarified that an observation is only identified by the URI of the observed resource and the identity of the client (#66).
- Clarified rules for establishing observation relationships (#68).
- Clarified conditions under which an observation relationship is terminated.
- Added explanation on how clients can terminate an observation relationship before the lifetime ends (#34).
- Clarified that the overriding objective for notifications is eventual consistency of the actual and the observed state (#67).
- Specified how a server needs to deal with clients not acknowledging confirmable messages carrying notifications (#69).
- Added a mechanism to detect message reordering (#35).
- Added an explanation of how notifications can be cached, supporting both the freshness and the validation model (#39, #64).
- Clarified that non-GET requests do not affect observation relationships, and that GET requests without "Lifetime" Option affecting relationships is by design (#65).
- Described interaction with blockwise transfers (#36).
- Added Resource Discovery section (#99).
- Added IANA Considerations.
- Added Security Considerations (#40).
- Added examples (#38).

Author's Address

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

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