CoRE Working Group
Internet-Draft
Intended status: Standards Track
Expires: April 25, 2013

K. Hartke
Universitaet Bremen TZI

October 22, 2012

Observing Resources in CoAP
draft-ietf-core-observe-07

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 a server to replicate a resource state to interested clients. The protocol follows a best-effort approach when transmitting new resource states to clients, and provides eventual consistency between the state observed by each client and the actual resource state.

Editor's Note

This is an interim revision which will receive further modifications during the resolution of open tickets, in particular #204, #235 and #242.

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 April 25, 2013.

Table of Contents

1. Introduction

1.1. Background

CoAP [I-D.ietf-core-coap] is an Application Protocol for Constrained Nodes/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 communication model of REST is that of a client exchanging resource states with an origin server using representations. The origin server is the definitive source for representations of the resources in its namespace. A client interested in the state of a resource sends a request to the origin server; the server then returns a response with a representation that is current at the time of the request.

This model does not work well when a client is interested in knowing the state of a resource over a period of time. Existing approaches when using HTTP, such as repeated polling or long-polls [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 to replicate a resource state from a server to interested clients over a period of time, while still keeping the properties of REST.

There is no intention for this mechanism 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, an observer must register separately for all of them. The pattern is typically used when a clean separation between related components is required, such as data storage and user interface.

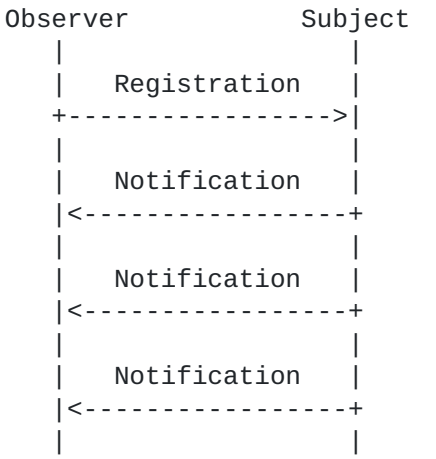

Figure 1: Observer Design Pattern

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 the current state of the resource at any given time.

Registration: A client registers its interest by sending 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 for the resource.

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

Figure 2 shows an example of a CoAP client registering its interest in a resource and receiving three notifications: the first with the current state upon registration and then two notifications when the state of the resource changes. Registration request and notifications are identified by the presence of the Observe Option defined in this document. All notifications echo the token specified by the client in the request, so the client can easily correlate them to the request.

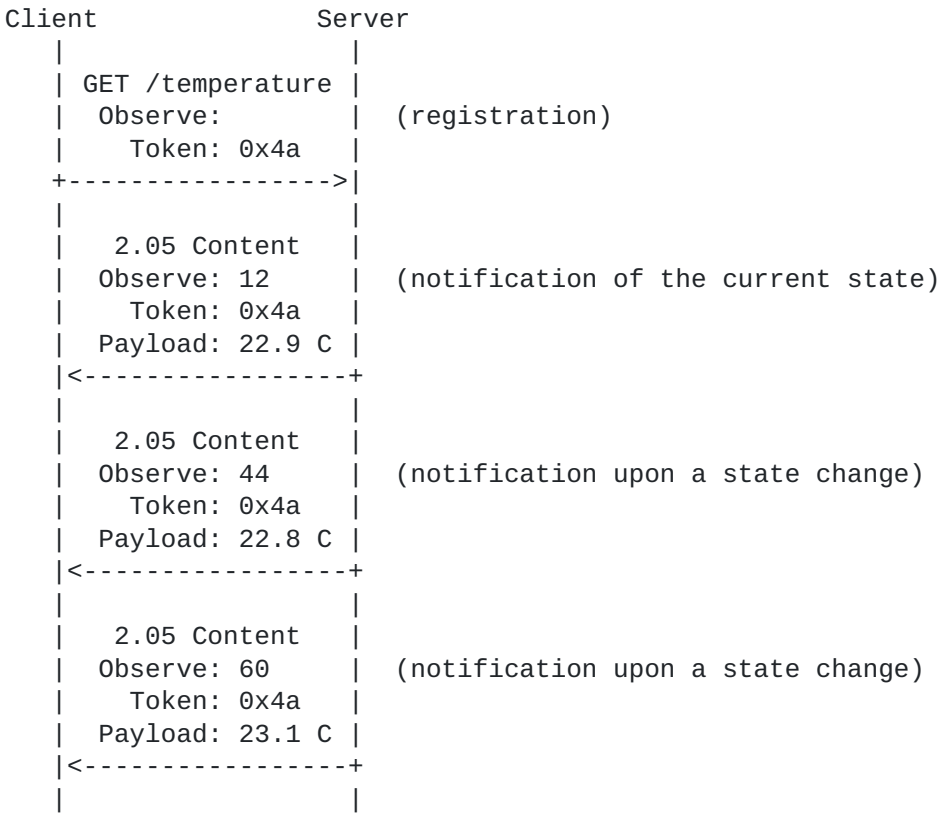

Figure 2: Observing a Resource in CoAP

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 by the server from the client's acknowledgement of notifications sent in confirmable messages. If the client 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 by the server from the list of observers.

A notification is cacheable like any other response and can be used until the next notification arrives. Each notification includes an indication of when the server will send the next notification at latest (Max-Age). This helps a client that does not receive a notification for a while, to decide if the resource simply did not undergo a change of state yet or if the next notification is overdue and the server is apparently no longer aware of the client's interest in the resource.

When a client wants to be notified after it has determined that no further notifications can be expected, it needs to register again.

1.3. Design Philosophy

The protocol builds on the architectural elements of REST, which include: a server that is responsible for the state and representation of the resources in its namespace, a client that is responsible for keeping the application state, and the stateless exchange of resource representations. (Beyond stateless REST, a server needs to keep track of the observers though, somewhat similar to how HTTP servers need to keep track of the TCP connections from their clients.) The protocol enables high scalability and efficiency through the support of caches and intermediaries that multiplex the interest of multiple clients in the same resource into a single association.

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 meaningful in the application context. Resources can be parameterized to achieve similar effects though; see Appendix B for examples.

Since bandwidth is in short supply in constrained environments, a server must adapt the rate of notifications to each client. This implies that a client cannot rely on observing every single state a resource goes through. Instead, the protocol follows a best-effort approach when transmitting the new resource state 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.

Furthermore, the protocol is designed on the principle of eventual consistency: it guarantees that if the resource does not undergo a new change in state, eventually all registered observers will have a current representation of the last resource state.

1.4. 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	x		Observe	empty/uint	0 B/0-2 B	(none)

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

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

In a response, the Observe Option identifies the message as a notification, which implies that the client has been added to the list of observers and that the server will notify the client of further changes to the resource state. The option's value is a sequence number that can be used for reordering detection (see Section 3.4 and Section 4.4). The value is encoded as a variable-length unsigned integer as defined in Section 3.4.1 of RFC XXXX [I-D.ietf-core-coap].

Since the Observe Option is not critical, a GET request that includes the Observe Option will automatically fall back to a normal GET request if the server is unwilling or unable to add the client to the list of observers.

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 for as long as the server can assume the client's interest.

3.2. Notifications

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

A notification can be confirmable or non-confirmable (i.e. sent in a confirmable or non-confirmable message). If a client does not recognize the token in a confirmable notification, it MUST NOT acknowledge the message and SHOULD reject it with a RST message. Otherwise, the client MUST acknowledge the message with an ACK message as usual. If a client does not recognize the token in a non-confirmable notification, it MAY reject it with a RST message.

An acknowledgement 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 it from the list of observers.

Notifications will have a 2.05 (Content) response code in most cases. They may also have a 2.03 (Valid) response code if the client includes an ETag Option in its request (see Section 3.3). In the event that the state of an observed resource is changed in a way that would cause a normal GET request not to return success (for example, when the resource is deleted), the server will send a notification with a non-success response code (such as 4.xx/5.xx) and empty the list of observers of the resource.

3.3. Caching

As notifications are just additional responses, notifications partake in caching as defined by Section 5.6 of RFC XXXX [I-D.ietf-core-coap]. Both the freshness model and the validation model are supported. The freshness model also serves as the model for the client to determine if it's still on the list of observers or if it needs to re-register its interest in the resource.

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

The server will do its best to keep the client up to date with a fresh representation of the current resource state. It will send a

notification whenever the resource changes, or at latest when the age of the last notification becomes greater than its Max-Age. (Note that this notification may not arrive in time due to network latency.)

The client SHOULD assume that it's on the list of observers while the age of the last notification is not greater than Max-Age. If the client does not receive a notification before the age becomes greater than Max-Age, it can assume that it has been removed from the list of observers (e.g., due to a loss of server state). In this case, it may need to re-register its interest.

To make sure it has a fresh representation and/or to re-register its interest, a client MAY issue a new GET request with an Observe Option at any time. The GET request SHOULD specify a new token to avoid ambiguity, because the token serves as 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 and, beyond that, while a new notification from the server is still likely to arrive. I.e. the client should wait until the age of the last notification becomes greater than its Max-Age plus MAX_LATENCY (the maximum time a datagram is expected to take from the start of its transmission to the completion of its reception; see Section 4.8 of RFC XXXX [I-D.ietf-core-coap]).

When a client has one or more notifications stored, it can use the ETag Option in the GET request to give the server an opportunity to select a stored response to be used. The client MAY include an ETag Option for each stored response that is applicable. It needs to keep those responses in the cache until it is no longer interested in receiving notifications for the target resource or it issues a new GET request with a new set of entity-tags. Whenever the observed resource changes its state 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.

3.4. Reordering

Messages that carry notifications can arrive in a different order than they were sent. Since the goal is eventual consistency (see Section 1.3), a client MAY safely skip a notification that arrives later than a newer notification. For this purpose, the server sets the value of the Observe Option in each notification to a 24-bit sequence number.

A notification is older than the latest notification received and thus can be skipped when the following condition is met:

(V1 - V2) % (2**24) < (2**23) and T2 < (T1 + EXCHANGE_LIFETIME)

where V1 is the value of the Observe Option of the latest valid notification received, V2 the value of the Observe Option of the present notification, T1 a client-local timestamp of the latest valid notification received (in seconds), and T2 a client-local timestamp of the present notification.

Design Note: The first condition essentially verifies that V2 > V1 holds in 24-bit sequence number arithmetic [RFC1982]. The second condition checks that the time expired between the two incoming messages is not so large that the sequence number might have wrapped around and the first check is therefore invalid. (In other words, after about EXCHANGE_LIFETIME seconds elapse without any notification, the client does not need to check the sequence numbers in order to assume an incoming notification is new.) The constant of 2**23 is non-critical, as is the even speed or precision of the clock involved.

3.5. Cancellation

When a client rejects a confirmable notification with a RST message or when it performs a GET request without an Observe Option for a currently observed resource, the server will remove the client from the list of observers for this resource. The client MAY use either method at any time to indicate that it is no longer interested in receiving notifications about a resource.

When a client rejects non-confirmable notification with a RST, there is also a chance that the server will remove the client from the list of observers for this resource. So the client MAY try this method as well. A client MAY rate-limit the RST 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 inadvertantly cancel an observation relationship by sending an unrelated GET to the same resource. To avoid this, without incurring a need for synchronization, such clients can use a different source transport address 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 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 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 state and, e.g., needs to poll the resource 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 same resource that does not include an Observe Option or a GET request that includes an unrecognized critical option, the server MUST remove the client 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 match. Message IDs and tokens are not taken into account.

Any request with a method other than GET MUST NOT have a direct effect on a list of observers of a resource. However, such a request can have the indirect consequence of causing the server to send a non-success notification which does affect the list of observers (e.g., when a DELETE request is successful and an observed resource no longer exists).

4.2. Notifications

A client is notified of resource state changes by additional responses sent by the server in reply to the GET request. Each such notification 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 to return a non-success response code (for example, if the resource is deleted), the server SHOULD notify the client by sending a notification with an appropriate non-success response code (such as 4.xx/5.xx) and MUST empty 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 empty the list of observers of the resource.

A notification can be sent as 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 implies the client's continued interest in being notified. If the client rejects a confirmable notification with a RST message, the server MUST remove the client from the list of observers. If the client rejects a non-confirmable notification with a RST message, the server MAY remove the client from the list of observers, i.e., it is expected that the server removes the client if it still has the state available that is needed to match the RST message to the notification, but the server is not required to keep this state.

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.

4.3. Caching

The Max-Age Option of a notification SHOULD be set to a value that indicates when the server will send the next notification. For example, if the server sends a notification every 30 seconds, a Max-Age Option with value 30 should be included. The server MAY send a new notification before Max-Age ends and MUST send a new notification at latest when Max-Age ends. If the client does not receive a new notification before Max-Age ends, it will assume that it was removed from the list of observers (e.g., due to a loss of server state) and may issue a new GET request to re-register its interest.

It may not always be possible to predict when the server will send the next notification, for example, when a resource does not change its state in regular intervals. In this case, the server SHOULD set Max-Age to a good approximation. The value is a trade-off between increased usage of bandwidth and the risk of stale information. Smaller values lead to more notifications and more GET requests, while greater values result in network or device failures being detected later and data becoming stale.

The client can include a set of entity-tags in its request using the ETag Option. When the 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 recipient has any response stored other than those identified by the entity-tags in the most recent GET request 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 in each notification to the 24 least-significant bits of a strictly increasing sequence number. The sequence number MAY start at any value. The server MUST NOT reuse the same option value with the same client, token and resource within approximately 2*EXCHANGE_LIFETIME seconds (roughly 8.5 minutes with default CoAP parameters).

Implementation Note: A simple implementation that satisfies the requirements is to use a timestamp (in seconds) provided by the device's clock, or a 24-bit unsigned integer variable that is incremented periodically and does not wrap around more often than every 2*EXCHANGE_LIFETIME seconds. It is not necessary that the clock reflects the correct local time or that it ticks in a precisely periodical way.

The client is comparing sequence numbers only between notifications that arrive within EXCHANGE_LIFETIME seconds. However, a server should not assume that it is free to completely forget the sequence number right after EXCHANGE_LIFETIME -- the client may have a slower clock. If there is a need to discard sequence number state after some inactivity, this should be done only after 2*EXCHANGE_LIFETIME or later. Similarly, the sequence number should not increase so fast to span more than half the sequence number space within less than 2*EXCHANGE_LIFETIME.

The 24-bit size for the sequence number has been chosen to enable very fast notification: If EXCHANGE_LIFETIME is the default value and 2*EXCHANGE_LIFETIME therefore approximately 2**9 seconds, the sequence number can increase every 2**14 times per second or approximately every 61 us before there is any danger of misdetection of wrap-around. On average, a server therefore is limited to about 15000 notifications per second per client and resource. This number may seem high in today's constrained node/ networks, but it allows some leeway for both increased EXCHANGE_LIFETIME and high notification frequencies.

4.5. Retransmission

In CoAP, confirmable messages are retransmitted in exponentially increasing intervals for a certain number of attempts until they are acknowledged by the client. In the context of observing a resource, it is undesirable to continue transmitting the representation of a resource state when the state has changed in the meantime.

When a server is in the process of delivering a confirmable notification and is waiting for an acknowledgement, and it wants to notify the client of a state change using a new confirmable message, it MUST stop retransmitting the old notification and SHOULD attempt to deliver the new notification with the number of attempts remaining from the old notification. When the last attempt to retransmit a confirmable message with a notification for a resource times out, the server SHOULD remove the client from the list of observers and additionally MAY remove the client from the lists of observers of all resources in its namespace.

The server SHOULD use a number of retransmit attempts (MAX_RETRANSMIT) such that removing a client from the list of observers before Max-Age ends is avoided.

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.

When a notification is transmitted multiple times (either as caused by a retransmission attempt or repeating it), the server MUST update value of the Observe Option. Otherwise, the client might discard the notification as too old.

5. Intermediaries

A client may be interested in a resource in the namespace of an origin server that is reached through 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. 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, i.e. each hop in the server role MUST determine individually how many notifications to send, of which 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 in the list of observers of a resource at 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 instance, just to keep its own cache up to date.

See Appendix A.1 for examples.

6. Block-wise Transfers

Resources observed by clients may be larger than can be comfortably processed or transferred in one CoAP message. CoAP provides a block-wise transfer mechanism to address this problem [I-D.ietf-core-block]. The following rules apply to the combination of block-wise 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 block-wise 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 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 value. The server performs the block-wise 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, or blocks that have been received more than approximately 2**14 seconds apart.

See Appendix A.2 for an example.

7. Discovery

A web link [RFC5988] to a resource accessible by the CoAP protocol MAY indicate that the server encourages the observation of this resource by including the link target attribute "obs". This is particularly useful in link-format documents [RFC6690].

The presence of this attribute can, for example, be used to indicate, via a graphical representation in a user interface, that this resource is changing its value and is useful for monitoring. The presence of this attribute is not a promise, though, that the Observe Option can actually be used to perform this observation. A client may need to resort to polling the resource if the Observe Option is not returned in the reply to the GET request.

The "obs" attribute is used as a flag, and thus has no value component -- a value given for the attribute MUST NOT be given for this version of the specification and MUST be ignored if present. The attribute MUST NOT be given more than once for this version of the specification.

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 entries are 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 Castellani, Gilbert Clark, Esko Dijk, Thomas Fossati, Brian Frank, Cullen Jennings, Matthias Kovatsch, Salvatore Loreto, Charles Palmer and Zach Shelby for helpful comments and discussions that have shaped the document.

Klaus Hartke was funded by the Klaus Tschira Foundation.

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-12 (work in progress), October 2012.

[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>.

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

[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

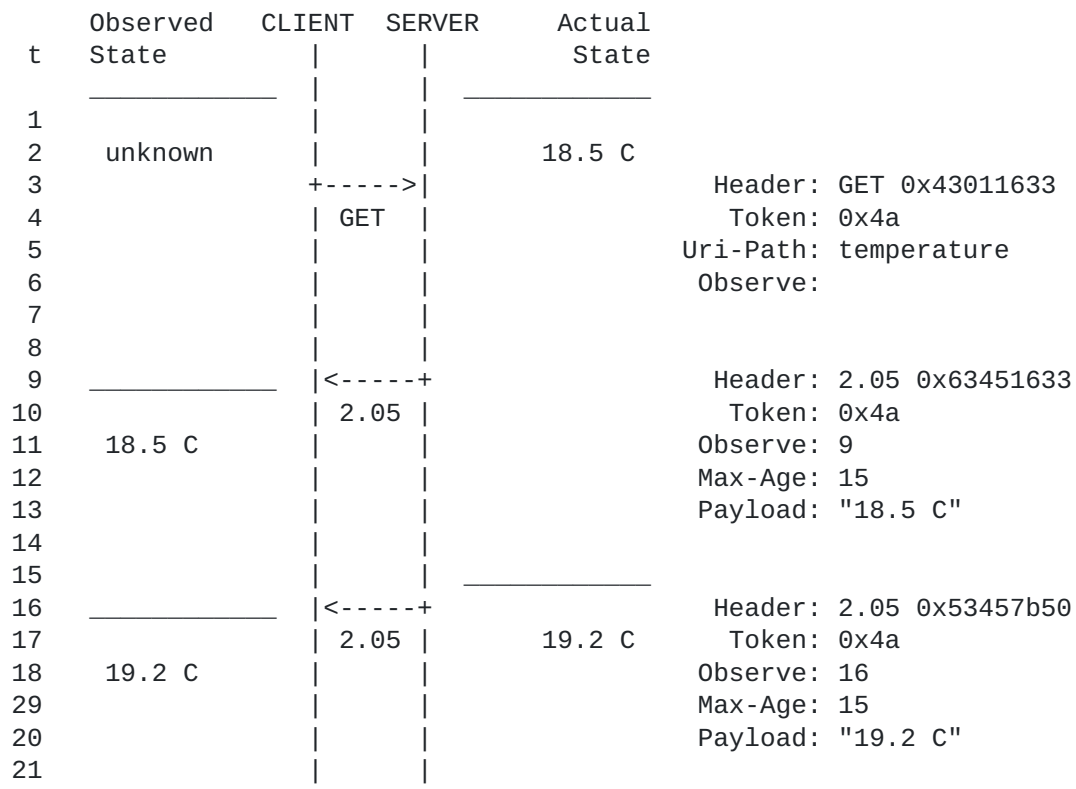

Figure 3: A client registers and receives a notification of the current state and upon a state change

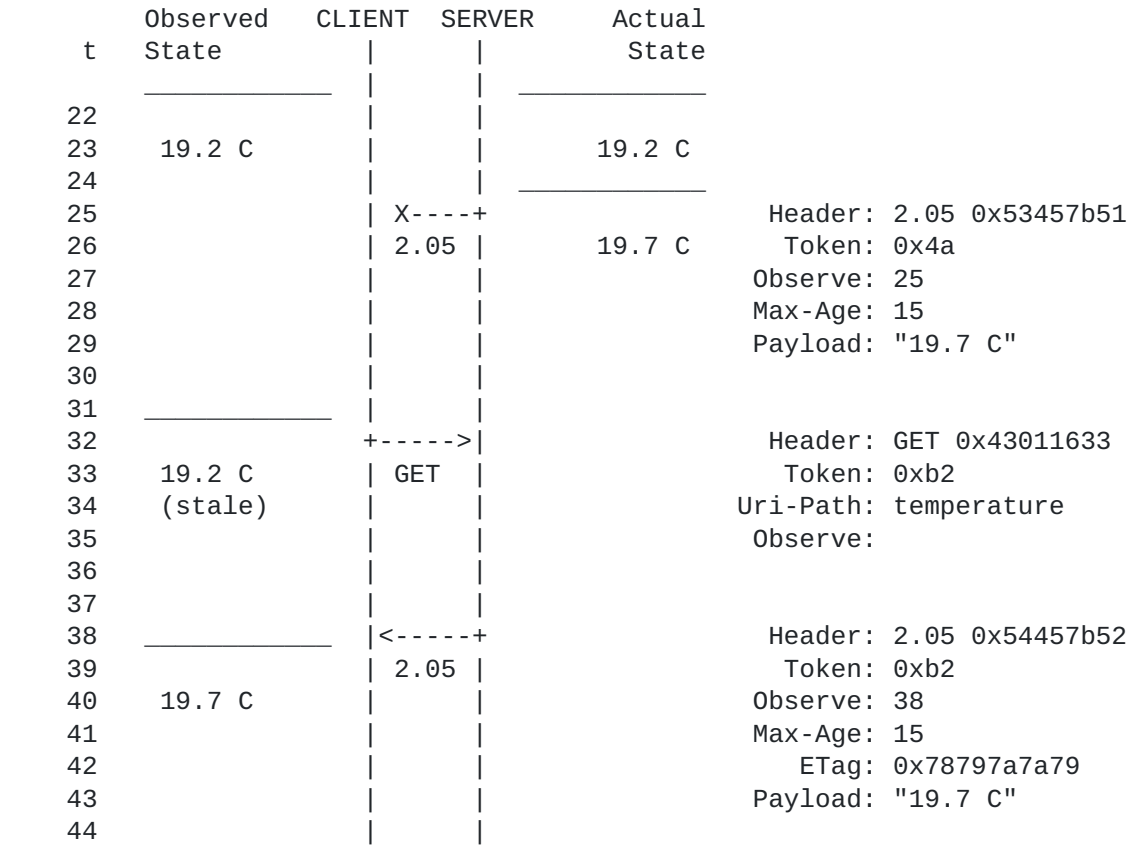

Figure 4: The client re-registers after Max-Age ends

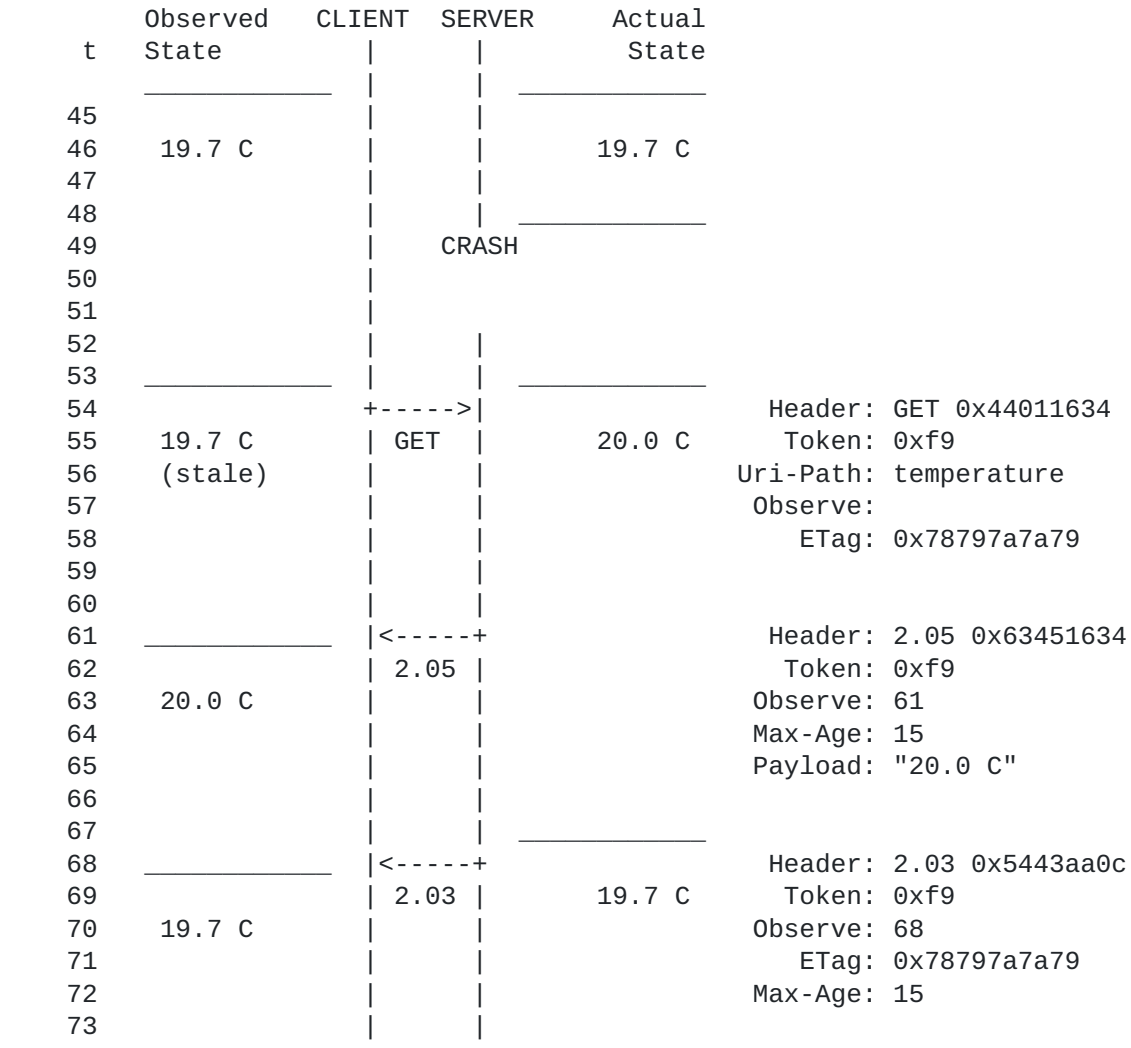

Figure 5: The client re-registers and gives the server the opportunity to select a stored response

A.1. Proxying

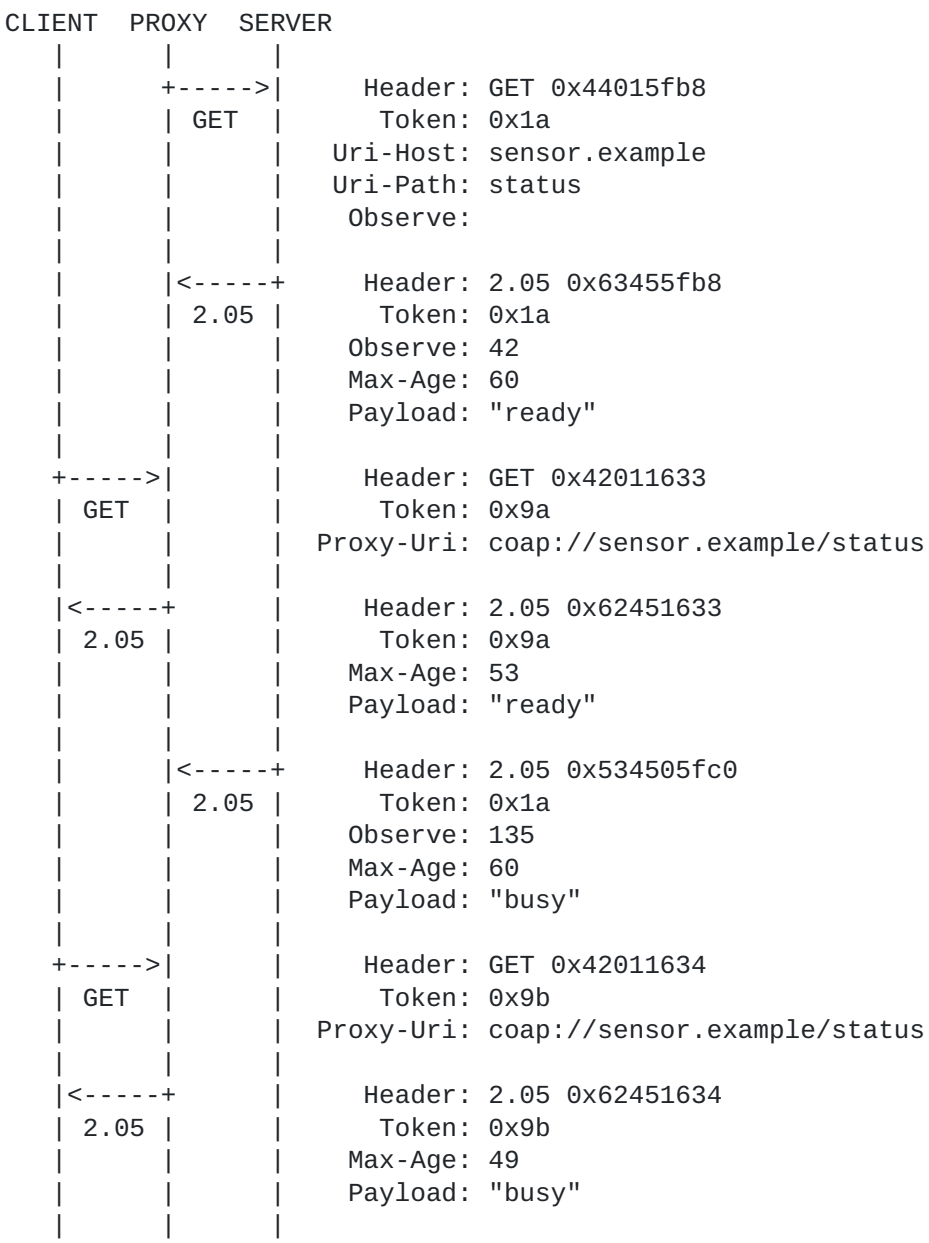

Figure 6: A proxy observes a resource to keep its cache up to date

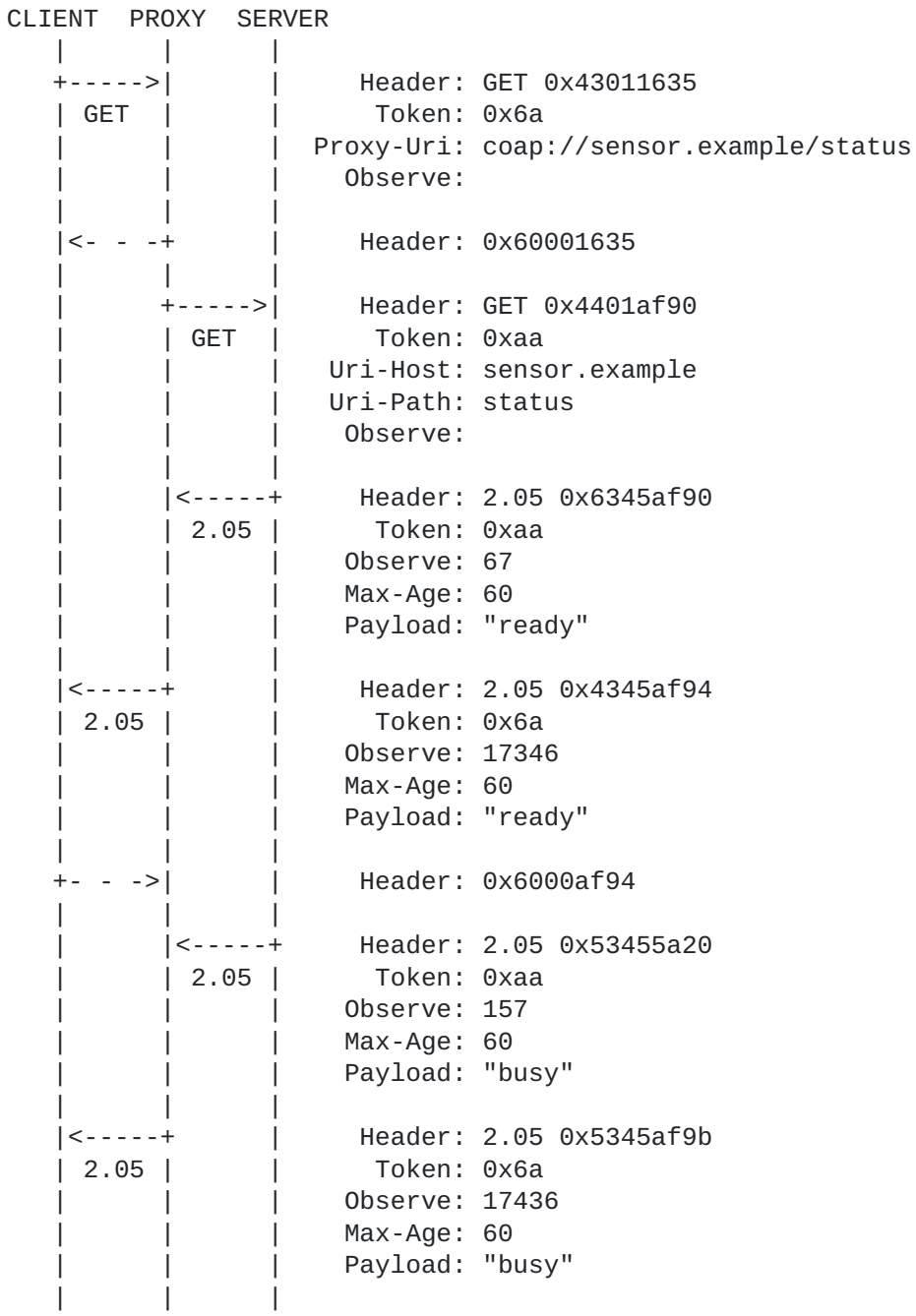

Figure 7: A client observes a resource through a proxy

A.2. Block-wise Transfer

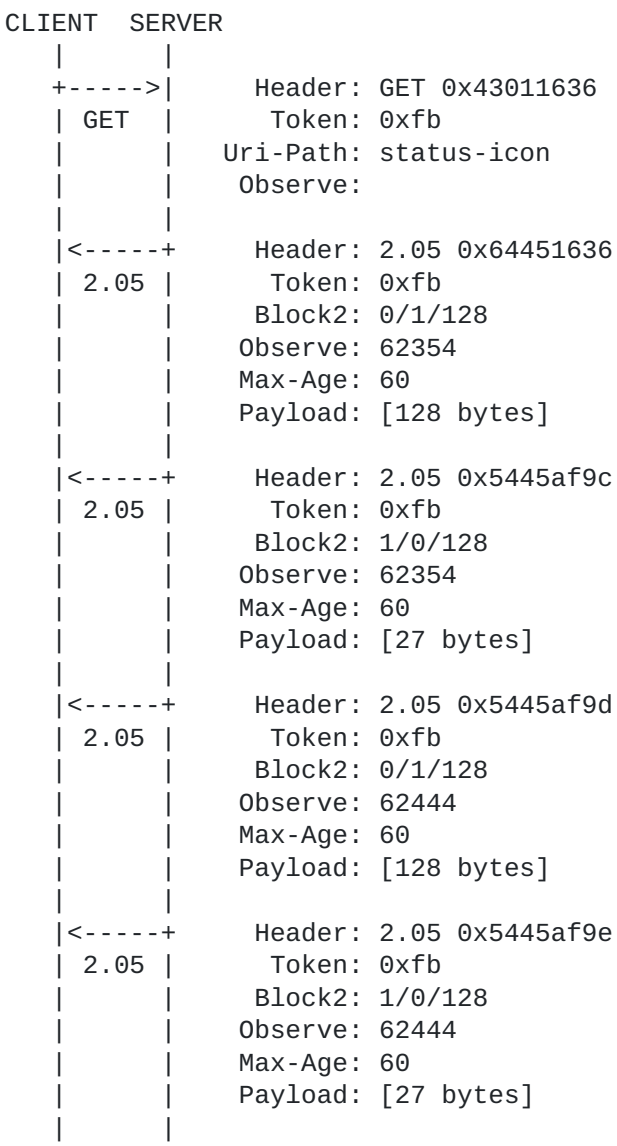

Figure 8: A server sends two notifications of two blocks each

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 the temperature drops below a preconfigured threshold, and to "warm" when the temperature exceeds a second, higher threshold;

- the server could expose a parameterized resource <coap://server/temperature/critical?above=45> that changes its state to the current temperature if the temperature exceeds the specified value, and 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. With parametrized 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-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 RST message that is 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 RST in reply to non-confirmable notifications.
- Added an implementation note about careless GETs (#184).
- Updated examples.

Changes from ietf-03 to ietf-04:

- Removed the "Max-OFE" Option.
- Allowed RST 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 block-wise 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 block-wise 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