Internet Engineering Task Force (IETF)
Request for Comments: 7641
Category: Standards Track
ISSN: 2070-1721
K. Hartke
Universitaet Bremen TZI
September 2015

Observing Resources in the Constrained Application Protocol (CoAP)

Abstract

The Constrained Application Protocol (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.

Status of This Memo

This is an Internet Standards Track document.

This document is a product of the Internet Engineering Task Force (IETF). It represents the consensus of the IETF community. It has received public review and has been approved for publication by the Internet Engineering Steering Group (IESG). Further information on Internet Standards is available in Section 2 of RFC 5741.

Information about the current status of this document, any errata, and how to provide feedback on it may be obtained at http://www.rfc-editor.org/info/rfc7641.

Copyright Notice

Copyright (c) 2015 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

1.1. Background

The Constrained Application Protocol (CoAP) [RFC7252] is intended to provide RESTful services [REST] not unlike HTTP [RFC7230] 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 [RFC7228].

The model of REST is that of a client exchanging representations of resources with a server, where a representation captures the current or intended state of a resource. The server is the authority 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 retrieves a representation of the resource and requests this representation be updated by the server as long as the client is interested in the resource.

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, the observer must register separately for all of them.

```
Observer            Subject
   |                   |
   |    Registration   |
   +------------------>|
   |                   |
   |    Notification   |
   |<------------------+
   |                   |
   |    Notification   |
   |<------------------+
   |                   |
   |    Notification   |
   |<------------------+
   |                   |
```

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 single extended 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 with the current state upon registration, and then two upon 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 additionally provides a sequence number for reordering detection. All notifications carry the token specified by the client, so the client can easily correlate them to the request.

```
      Client                     Server
         |                          |
         |  GET /temperature        |
         |    Token: 0x4a           |   Registration
         |  Observe: 0              |
         +------------------->|
         |                          |
         |    2.05 Content          |
         |    Token: 0x4a           |   Notification of
         |  Observe: 12             |   the current state
         |  Payload: 22.9 Cel       |
         |<-------------------+
         |                          |
         |    2.05 Content          |
         |    Token: 0x4a           |   Notification upon
         |  Observe: 44             |   a state change
         |  Payload: 22.8 Cel       |
         |<-------------------+
         |                          |
         |    2.05 Content          |
         |    Token: 0x4a           |   Notification upon
         |  Observe: 60             |   a state change
         |  Payload: 23.1 Cel       |
         |<-------------------+
         |                          |
```

Figure 2: Observing a Resource in CoAP

Note: In this document, "Cel" stands for "degrees Celsius".

A client remains on the list of observers as long as the server can determine the client's continued interest in the resource. The server may send a notification in a confirmable CoAP message to request an acknowledgement from the client. When the client deregisters, rejects a notification, or the transmission of a notification times out after several transmission attempts, the client is considered no longer interested in the resource and is removed by the server from the list of observers.

1.3. Consistency Model

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

It cannot be avoided that the client and the server become out of sync at times: First, there is always some latency between the change of the resource state and the receipt of the notification. Second, CoAP messages with notifications can get lost, which will cause the client to assume an old state until it receives a new notification. And third, the server may erroneously come to the conclusion that the client is no longer interested in the resource, which will cause the server to stop sending notifications and the client to assume an old state until it eventually 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. This is limited by congestion control, however, so a client cannot rely on observing every single state that a resource might go 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 limit, the client cannot use the enclosed representation until it receives 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.

1.4. Observable Resources

A CoAP server is the authority for determining under what conditions resources change their state and thus when observers are notified of new resource states. The protocol does not offer explicit means for setting up triggers or thresholds; it is up to the server to expose observable resources that change their state in a way that is useful in the application context.

For example, a CoAP server with an attached temperature sensor could expose one or more of the following resources:

- <coap://server/temperature>, which changes its state every few seconds to a current reading of the temperature sensor;

- <coap://server/temperature/felt>, which changes its state to "COLD" whenever the temperature reading drops below a certain pre-configured threshold and to "WARM" whenever the reading exceeds a second, slightly higher threshold;

- <coap://server/temperature/critical?above=42>, which changes its state based on the client-specified parameter value either every few seconds to the current temperature reading if the temperature exceeds the threshold or to "OK" when the reading drops below;

- <coap://server/?query=select+avg(temperature)+from+Sensor.window:time(30sec)>, which accepts expressions of arbitrary complexity and changes its state accordingly.

Thus, by designing CoAP resources that change their state on certain conditions, it is possible to update the client only when these conditions occur instead of supplying it continuously with raw sensor data. By parameterizing resources, this is not limited to conditions defined by the server, but can be extended to arbitrarily complex queries specified by the client. The application designer therefore can choose exactly the right level of complexity for the application envisioned and devices involved and is not constrained to a "one size fits all" mechanism built into the protocol.

1.5. 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

The Observe Option has the following properties. Its meaning depends on whether it is included in a GET request or in a response.

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

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

Table 1: The Observe Option

When included in a GET request, the Observe Option extends the GET method so it does not only retrieve a current representation of the target resource, but also requests the server to add or remove an entry in the list of observers of the resource depending on the option value. The list entry consists of the client endpoint and the token specified by the client in the request. Possible values are:

0 (register) adds the entry to the list, if not present;

1 (deregister) removes the entry from the list, if present.

The Observe Option is not critical for processing the request. If the server is unwilling or unable to add a new entry to the list of observers, then the request falls back to a normal GET request and the response does not include the Observe Option.

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.

When included in a response, the Observe Option identifies the message as a notification. This implies that a matching entry exists in the list of observers and that the server will notify the client of changes to the resource state. The option value is a sequence number for reordering detection (see Sections 3.4 and 4.4).

The value of the Observe Option is encoded as an unsigned integer in network byte order using a variable number of bytes ('uint' option format); see Section 3.2 of RFC 7252 [RFC7252].

3. Client-Side Requirements

3.1. Request

A client registers its interest in a resource by issuing a GET request with an Observe Option set to 0 (register). If the server returns a 2.xx response that includes an Observe Option as well, the server has successfully added an entry with the client endpoint and request token to the list of observers of the target resource, and the client will be notified of changes to the resource state.

Like a fresh response can be used to satisfy a request without contacting the server, the stream of updates resulting from one observation request can be used to satisfy another (observation or normal GET) request if the target resource is the same. A client MUST aggregate such requests and MUST NOT register more than once for the same target resource. The target resource is identified by all options in the request that are part of the Cache-Key. This includes, for example, the full request URI and the Accept Option.

3.2. Notifications

Notifications are additional responses sent by the server in reply to the single extended GET request that created the registration. Each notification includes the token specified by the client in the request. The only difference between a notification and a normal response is the presence of the Observe Option.

Notifications typically have a 2.05 (Content) response code. They include 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. If the client included one or more ETag Options in the GET request (see Section 3.3), notifications can have a 2.03 (Valid) response code rather than a 2.05 (Content) response code. Such notifications include an Observe Option with a sequence number but no payload.

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 an appropriate response code (such as 4.04 Not Found) and removes the client's entry from the list of observers of the resource. Non-2.xx responses do not include an Observe Option.

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 7252 [RFC7252]. 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 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 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 any number of 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 assume that the enclosed representation reflects the actual resource state.

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 the same token as the original at any time. All options MUST be identical to those in the original request except for the set of ETag Options. It is RECOMMENDED that the client does not issue the request while it still has a fresh notification/response for the resource in its cache. Additionally, the client SHOULD at least wait for a random amount of time between 5 and 15 seconds after Max-Age expired to reduce collisions 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.

A client implementation needs to keep all candidate responses in its cache until it is no longer interested in the target resource or it re-registers 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 MUST consider the notification that was sent most recently as the freshest, regardless of the order of arrival.

To provide an order among notifications for the client, the server sets the value of the Observe Option in each notification to the 24 least significant bits of a strictly increasing sequence number. An incoming notification was sent more recently than the freshest notification 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 in the freshest notification so far, V2 is the value of the Observe Option in the incoming notification, T1 is a client-local timestamp for the freshest notification so far, and T2 is a client-local timestamp for the incoming notification.

Design Note: The first two conditions verify that V1 is less than V2 in 24-bit serial number arithmetic [RFC1982]. The third condition ensures that if the server is generating serial numbers based on a local clock, 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 serial number; in other words, after 128 seconds have elapsed without any notification, a client does not need to check the sequence numbers to assume that an incoming notification was sent more recently than the freshest notification it has received so far.

The duration of 128 seconds was chosen as a nice round number greater than MAX_LATENCY (Section 4.8.2 of RFC 7252 [RFC7252]).

3.5. Transmission

A notification can be confirmable or non-confirmable, i.e., it can be sent in a confirmable or a non-confirmable message. The message type used for a notification is independent of the type used for the request and of 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 associated entry from the list of observers (Section 4.5).

3.6. Cancellation

A client that is no longer interested in receiving notifications for a resource can simply "forget" the observation. When the server then sends the next notification, the client will not recognize the token in the message and thus will return a Reset message. This causes the server to remove the associated entry from the list of observers. The entries in lists of observers are effectively "garbage collected" by the server.

Implementation Note: Due to potential message loss, the Reset message may not reach the server. The client may therefore have to reject multiple notifications, each with one Reset message, until the server finally removes the associated entry from the list of observers and stops sending notifications.

In some circumstances, it may be desirable to cancel an observation and release the resources allocated by the server to it more eagerly. In this case, a client MAY explicitly deregister by issuing a GET request that has the Token field set to the token of the observation to be cancelled and includes an Observe Option with the value set to 1 (deregister). All other options MUST be identical to those in the registration request except for the set of ETag Options. When the server receives such a request, it will remove any matching entry from the list of observers and process the GET request as usual.

4. Server-Side Requirements

4.1. Request

A GET request with an Observe Option set to 0 (register) requests the server not only to return a current representation of the target resource, but also to add the client to the list of observers of that resource. Upon success, the server returns a current representation of the resource and MUST keep this representation updated (as described in Section 1.3) as long as the client is on the list of observers.

The entry in the list of observers is keyed by the client endpoint and the token specified by the client in the request. If an entry with a matching endpoint/token pair is already present in the list (which, for example, happens when the client wishes to reinforce its interest in a resource), the server MUST NOT add a new entry but MUST replace or update the existing one.

A server that is unable or unwilling to add a new entry to the list of observers of a resource MAY silently ignore the registration request 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 Observe Option in a GET request is set to 1 (deregister), then the server MUST remove any existing entry with a matching endpoint/token pair from the list of observers and process the GET request as usual. The resulting response MUST NOT include an Observe Option.

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 echo the token specified by the client in the GET request. If there are multiple entries in the list of observers, the order in which the clients 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 an appropriate response code (such as 4.04 Not Found) and subsequently MUST remove the associated entry from the list of observers of the resource.

The Content-Format specified in a 2.xx 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 in this format, it SHOULD send a notification with a 4.06 (Not Acceptable) response code and subsequently MUST remove the associated entry from the list of observers of the resource.

A 2.xx notification MUST include an Observe Option with a sequence number as specified in Section 4.4 below; a non-2.xx notification MUST NOT include an Observe Option.

4.3. Caching

As notifications are just additional responses sent by the server in reply to a GET request, they are subject to caching as defined in Section 5.6 of RFC 7252 [RFC7252].

4.3.1. Freshness

After returning the initial response, the server MUST keep the resource state that is 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, the client cannot tell if the observed state and the actual state are still in sync. Thus, when the age of the latest notification becomes greater than its indicated Max-Age, the client no longer has a usable representation of the resource state. The server MAY wish to prevent that by sending a new notification with the unchanged representation and a new Max-Age just before the Max-Age indicated earlier expires.

4.3.2. Validation

A client can include a set of entity tags in its request using the ETag Option. When an 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.

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 2^23 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 with the same token for the same resource. The value of the Observe Option MUST be current at the time of transmission; if a notification is retransmitted, the server MUST update the value of the option to the sequence number that is current at that time before retransmission.

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)/(2^23) = 30.52 microseconds. It is not necessary that the clock reflects the current time/date.

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 2^23 times in the first 256 seconds after every state change). This removes the need to update the value of the Observe Option on retransmission when the resource state did not change.

Design Note: The choice of a 24-bit option value and a time span of 256 seconds theoretically allows for a notification rate of up to 65536 notifications per second. Constrained nodes often have rather imprecise clocks, though, and inaccuracies of the client and server side may cancel out or add in effect. Therefore, the maximum notification rate is reduced to 32768 notifications per second. This is still well beyond the highest known design objective of around 1 kHz (most CoAP applications will be several orders of magnitude below that) but allows total clock inaccuracies of up to -50/+100%.

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.

A server MAY choose to skip sending a notification if it knows that it will send another notification soon, for example, when the state of a resource is changing frequently. It also MAY choose to send more than one notification for the same resource state. However, above all, the server MUST ensure that a client in the list of observers of a resource eventually observes the latest state if the resource does not undergo a new change in state.

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 by repeating the last notification in a confirmable message when the burst is over.

The client's acknowledgement of a confirmable notification signals that the client is interested in receiving further notifications. If a client rejects a confirmable or non-confirmable notification with a Reset message, or if the last attempt to retransmit a confirmable notification times out, then the client is considered no longer interested and the server MUST remove the associated entry from the list of observers.

Implementation Note: To properly process a Reset message that rejects a non-confirmable notification, a server needs to remember the message IDs of the non-confirmable notifications it sends. This may be challenging for a server with constrained resources. However, since Reset messages are transmitted unreliably, the client must be prepared in case the Reset messages are not received by the server. Thus, a server can always pretend that a Reset message rejecting a non-confirmable notification was lost.

If a server does this, it could accelerate cancellation by sending the following notifications to that client in confirmable messages.

A server that transmits notifications mostly in non-confirmable messages MUST send a notification in a confirmable message instead of a non-confirmable message at least every 24 hours. This prevents a client that went away or is no longer interested from remaining in the list of observers indefinitely.

4.5.1. Congestion Control

Basic congestion control for CoAP is provided by the exponential back-off mechanism in Section 4.2 of RFC 7252 [RFC7252] and the limitations in Section 4.7 of RFC 7252 [RFC7252]. However, CoAP places the responsibility of congestion control for simple request/response interactions only on the clients: rate-limiting request transmission implicitly controls the transmission of the responses. When a single request yields a potentially infinite number of notifications, additional responsibility needs to be placed on the server.

In order not to cause congestion, servers MUST strictly limit the number of simultaneous outstanding notifications/responses that they transmit to a given client to NSTART (1 by default; see Section 4.7 of RFC 7252 [RFC7252]). An outstanding notification/response is either a confirmable message for which an acknowledgement has not yet been received and whose last retransmission attempt has not yet timed out or a non-confirmable message for which the waiting time that results from the following rate-limiting rules has not yet elapsed.

The server SHOULD NOT send more than one non-confirmable notification per round-trip time (RTT) to a client on average. If the server cannot maintain an RTT estimate for a client, it SHOULD NOT send more than one non-confirmable notification every 3 seconds and SHOULD use an even less aggressive rate when possible (see also Section 3.1.2 of RFC 5405 [RFC5405]).

Further congestion control optimizations and considerations are expected in the future with advanced CoAP congestion control mechanisms.

4.5.2. Advanced Transmission

The state of an observed resource may change while the number of simultaneous outstanding notifications/responses to a client on the list of observers is greater than or equal to NSTART. In this case, the server cannot notify the client of the new resource state immediately but has to wait for an outstanding notification/response to complete first.

If there exists an outstanding notification/response that the server transmits to the client and that pertains to the changed resource, then it is desirable for the server to stop working towards getting the representation of the old resource state to the client and to start transmitting the current representation to the client instead, so the resource state observed by the client stays closer in sync with the actual state at the server.

For this purpose, the server MAY optimize the transmission process by aborting the transmission of the old notification (but not before the current transmission attempt is completed) and starting a new transmission for the new notification (but with the retransmission timer and counter of the aborted transmission retained).

In more detail, a server MAY supersede an outstanding transmission that pertains to an observation as follows:

1. Wait for the current (re)transmission attempt to be acknowledged, rejected, or to time out (confirmable transmission); or, wait for the waiting time to elapse or the transmission to be rejected (non-confirmable transmission).

2. If the transmission is rejected or it was the last attempt to retransmit a notification, remove the associated entry from the list of observers of the observed resource.

3. If the entry is still in the list of observers, start to transmit a new notification with a representation of the current resource state. Should the resource have changed its state more than once in the meantime, the notifications for the intermediate states are silently skipped.

4. The new notification is transmitted with a new Message ID and the following transmission parameters: if the previous (re)transmission attempt timed out, retain its transmission parameters, increment the retransmission counter, and double the timeout; otherwise, initialize the transmission parameters as usual (see Section 4.2 of RFC 7252 [RFC7252]).

It is possible that the server later receives an acknowledgement for a confirmable notification that it superseded this way. Even though this does not signal consistency, it is valuable in that it signals the client's further interest in the resource. The server therefore should avoid inadvertently removing the associated entry from the list of observers.

5. Intermediaries

A client may be interested in a resource in the namespace of a server that is reached through a chain of one or more CoAP intermediaries. In this case, the client registers its interest with the first intermediary towards the server, acting as if it was communicating with the 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 to keep the representation updated upon changes to the resource state, 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 server. If the response returned by the next hop doesn't include 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 in notifications and MUST set the value of the Max-Age Option according to the age of the local current representation.

If two or more clients have registered their interest in a resource with an intermediary, the intermediary MUST register itself only once with the next hop and fan out the notifications it receives to all registered clients. This relieves the next hop from sending the same notifications multiple times and thus enables scalability.

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.2 for examples.

6. 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.

7. Security Considerations

The security considerations in Section 11 of [RFC7252], the CoAP specification, apply.

Observing resources can dramatically increase the negative effects of amplification attacks. That is, not only can notifications messages be much larger than the request message, but the nature of the protocol can cause a significant number of notifications to be generated. Without client authentication, a server therefore MUST 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. Note that an attacker may still spoof the acknowledgements if the confirmable messages are sufficiently predictable.

The protocol follows a best-effort approach for keeping the state observed by a client and the actual resource state at a server in sync. This may have the client and the server become out of sync at times. Depending on the sensitivity of the observed resource, operating on an old state might be a security threat. The client therefore must be careful not to use a representation after its Max-Age expires, and the server must set the Max-Age Option to a sensible value.

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 apply access controls to 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.

Resources can be observed over CoAP that is secured by Datagram Transport Layer Security (DTLS) using any of the security modes described in Section 9 of RFC 7252. The use of DTLS is indicated by the "coaps" URI scheme. All notifications resulting from a GET request with an Observe Option MUST be returned within the same epoch of the same connection as the request.

8. IANA Considerations

The following entry has been added to the CoAP Option Numbers registry:

Number	Name	Reference
6	Observe	RFC 7641

9. References

9.1. Normative References

[RFC2119] Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, DOI 10.17487/RFC2119, March 1997, <http://www.rfc-editor.org/info/rfc2119>.

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

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

9.2. Informative References

[GOF] Gamma, E., Helm, R., Johnson, R., and J. Vlissides, "Design Patterns: Elements of Reusable Object-Oriented Software", Addison-Wesley Professional Computing Series, 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, DOI 10.17487/RFC1982, August 1996, <http://www.rfc-editor.org/info/rfc1982>.

[RFC5405] Eggert, L. and G. Fairhurst, "Unicast UDP Usage Guidelines for Application Designers", BCP 145, RFC 5405, DOI 10.17487/RFC5405, November 2008, <http://www.rfc-editor.org/info/rfc5405>.

[RFC5989] Roach, A., "A SIP Event Package for Subscribing to Changes to an HTTP Resource", RFC 5989, DOI 10.17487/RFC5989, October 2010, <http://www.rfc-editor.org/info/rfc5989>.

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

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

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

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

Appendix A. Examples

A.1. Client/Server Examples

```
         Observed   CLIENT  SERVER     Actual
     t   State         |      |          State
         ____________  |      |  ____________
     1                 |      |
     2    unknown      |      |     18.5 Cel
     3                 +----->|                  Header: GET 0x41011633
     4                 | GET  |                   Token: 0x4a
     5                 |      |                Uri-Path: temperature
     6                 |      |                 Observe: 0 (register)
     7                 |      |
     8                 |      |
     9   ____________  |<-----+                  Header: 2.05 0x61451633
    10                 | 2.05 |                   Token: 0x4a
    11    18.5 Cel     |      |                 Observe: 9
    12                 |      |                 Max-Age: 15
    13                 |      |                 Payload: "18.5 Cel"
    14                 |      |
    15                 |      |  ____________
    16   ____________  |<-----+                  Header: 2.05 0x51457b50
    17                 | 2.05 |     19.2 Cel      Token: 0x4a
    18    19.2 Cel     |      |                 Observe: 16
    29                 |      |                 Max-Age: 15
    20                 |      |                 Payload: "19.2 Cel"
    21                 |      |
```

Figure 3: A Client Registers and Receives One Notification of the Current State and One of a New State upon a State Change

```
         Observed   CLIENT  SERVER     Actual
     t   State         |      |         State
         ____________  |      |  ____________
    22                 |      |
    23    19.2 Cel     |      |     19.2 Cel
    24                 |      |  ____________
    25                 | X----+                  Header: 2.05 0x51457b51
    26                 | 2.05 |     19.7 Cel      Token: 0x4a
    27                 |      |                 Observe: 25
    28                 |      |                 Max-Age: 15
    29                 |      |                 Payload: "19.7 Cel"
    30                 |      |
    31   ____________  |      |
    32                 |      |
    33    19.2 Cel     |      |
    34    (stale)      |      |
    35                 |      |
    36                 |      |
    37                 |      |
    38                 +----->|                  Header: GET 0x41011634
    39                 | GET  |                   Token: 0xb2
    40                 |      |                Uri-Path: temperature
    41                 |      |                 Observe: 0 (register)
    42                 |      |
    43                 |      |
    44   ____________  |<-----+                  Header: 2.05 0x61451634
    45                 | 2.05 |                   Token: 0xb2
    46    19.7 Cel     |      |                 Observe: 44
    47                 |      |                 Max-Age: 15
    48                 |      |                    ETag: 0x78797a7a79
    49                 |      |                 Payload: "19.7 Cel"
    50                 |      |

           Figure 4: The Client Re-registers after Max-Age Ends
```

```
         Observed   CLIENT  SERVER     Actual
     t   State         |      |          State
         ____________  |      |  ____________
    51                 |      |
    52    19.7 Cel     |      |     19.7 Cel
    53                 |      |
    54                 |      |  ____________
    55                 |    crash
    56                 |
    57                 |
    58                 |
    59   ____________  |
    60                 |
    61    19.7 Cel     |
    62    (stale)      |
    63                 |   reboot____________
    64                 |      |
    65                 |      |     20.0 Cel
    66                 |      |
    67                 +----->|                  Header: GET 0x41011635
    68                 | GET  |                   Token: 0xf9
    69                 |      |                Uri-Path: temperature
    70                 |      |                 Observe: 0 (register)
    71                 |      |                    ETag: 0x78797a7a79
    72                 |      |
    73                 |      |
    74   ____________  |<-----+                  Header: 2.05 0x61451635
    75                 | 2.05 |                   Token: 0xf9
    76    20.0 Cel     |      |                 Observe: 74
    77                 |      |                 Max-Age: 15
    78                 |      |                 Payload: "20.0 Cel"
    79                 |      |
    80                 |      |  ____________
    81   ____________  |<-----+                  Header: 2.03 0x5143aa0c
    82                 | 2.03 |     19.7 Cel      Token: 0xf9
    83    19.7 Cel     |      |                 Observe: 81
    84                 |      |                    ETag: 0x78797a7a79
    85                 |      |                 Max-Age: 15
    86                 |      |
```

Figure 5: The Client Re-registers and Gives the Server the Opportunity to Select a Stored Response

```
         Observed   CLIENT  SERVER     Actual
     t   State         |      |         State
         ____________  |      |  ____________
    87                 |      |
    88    19.7 Cel     |      |     19.7 Cel
    89                 |      |
    90                 |      |  ____________
    91   ____________  |<-----+                  Header: 2.05 0x4145aa0f
    92                 | 2.05 |     19.3 Cel      Token: 0xf9
    93    19.3 Cel     |      |                 Observe: 91
    94                 |      |                 Max-Age: 15
    95                 |      |                 Payload: "19.3 Cel"
    96                 |      |
    97                 |      |
    98                 +- - ->|                  Header: 0x7000aa0f
    99                 |      |
   100                 |      |
   101                 |      |
   102                 |      |  ____________
   103                 |      |
   104                 |      |     19.0 Cel
   105                 |      |
   106   ____________  |      |
   107                 |      |
   108    19.3 Cel     |      |
   109    (stale)      |      |
   110                 |      |
```

    Figure 6: The Client Rejects a Notification and Thereby Cancels the
                                Observation

A.2. Proxy Examples

```
   CLIENT  PROXY  SERVER
     |       |       |
     |       +------>|     Header: GET 0x41015fb8
     |       | GET   |      Token: 0x1a
     |       |       |   Uri-Host: sensor.example
     |       |       |   Uri-Path: status
     |       |       |    Observe: 0 (register)
     |       |       |
     |       |<------+     Header: 2.05 0x61455fb8
     |       | 2.05  |      Token: 0x1a
     |       |       |    Observe: 42
     |       |       |    Max-Age: 60
     |       |       |    Payload: "ready"
     |       |       |
     +------>|       |     Header: GET 0x41011633
     | GET   |       |      Token: 0x9a
     |       |       |  Proxy-Uri: coap://sensor.example/status
     |       |       |
     |<------+       |     Header: 2.05 0x61451633
     | 2.05  |       |      Token: 0x9a
     |       |       |    Max-Age: 53
     |       |       |    Payload: "ready"
     |       |       |
     |       |<------+     Header: 2.05 0x514505fc0
     |       | 2.05  |      Token: 0x1a
     |       |       |    Observe: 135
     |       |       |    Max-Age: 60
     |       |       |    Payload: "busy"
     |       |       |
     +------>|       |     Header: GET 0x41011634
     | GET   |       |      Token: 0x9b
     |       |       |  Proxy-Uri: coap://sensor.example/status
     |       |       |
     |<------+       |     Header: 2.05 0x61451634
     | 2.05  |       |      Token: 0x9b
     |       |       |    Max-Age: 49
     |       |       |    Payload: "busy"
     |       |       |
```

Figure 7: A Proxy Observes a Resource to Keep its Cache Up to Date

```
   CLIENT  PROXY  SERVER
      |       |       |
      +------>|       |     Header: GET 0x41011635
      | GET   |       |      Token: 0x6a
      |       |       |  Proxy-Uri: coap://sensor.example/status
      |       |       |    Observe: 0 (register)
      |       |       |
      |<- - -+        |     Header: 0x60001635
      |       |       |
      |       +------>|     Header: GET 0x4101af90
      |       | GET   |      Token: 0xaa
      |       |       |   Uri-Host: sensor.example
      |       |       |   Uri-Path: status
      |       |       |    Observe: 0 (register)
      |       |       |
      |       |<------+     Header: 2.05 0x6145af90
      |       | 2.05  |      Token: 0xaa
      |       |       |    Observe: 67
      |       |       |    Max-Age: 60
      |       |       |    Payload: "ready"
      |       |       |
      |<------+       |     Header: 2.05 0x4145af94
      | 2.05  |       |      Token: 0x6a
      |       |       |    Observe: 17346
      |       |       |    Max-Age: 60
      |       |       |    Payload: "ready"
      |       |       |
      +- - ->|        |     Header: 0x6000af94
      |       |       |
      |       |<------+     Header: 2.05 0x51455a20
      |       | 2.05  |      Token: 0xaa
      |       |       |    Observe: 157
      |       |       |    Max-Age: 60
      |       |       |    Payload: "busy"
      |       |       |
      |<------+       |     Header: 2.05 0x5145af9b
      | 2.05  |       |      Token: 0x6a
      |       |       |    Observe: 17436
      |       |       |    Max-Age: 60
      |       |       |    Payload: "busy"
      |       |       |
```

Figure 8: A Client Observes a Resource through a Proxy

Acknowledgements

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

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

This work was supported in part by Klaus Tschira Foundation, Intel, Cisco, and Nokia.

Author’s Address

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

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