CoRE Working Group K. Hartke
Internet-Draft Universitaet Bremen TZI
Intended status: Standards Track Z. Shelby
Expires: September 16, 2011 Sensinode
March 15, 2011

Observing Resources in CoAP
draft-ietf-core-observe-02

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 specification provides a simple extension for CoAP that gives clients the ability to observe such changes.

Status of this Memo

Copyright Notice

Table of Contents

1. Introduction

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 state of a resource on a CoAP server can change over time. We want to give CoAP clients the ability to observe this change. However, existing approaches from the HTTP world, such as repeated polling or long-polls, generate significant complexity and/or overhead and thus are less applicable in the constrained CoAP world. Instead, a much simpler mechanism is provided to solve the basic problem of resource observation. Note that 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].

This specification describes an architecture and a protocol design that realizes the well-known subject/observer design pattern within the REST-based environment of CoAP.

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 [RFC2119].

Where arithmetic is explained, this document uses the notation familiar from the programming language C, except that the operator "^" stands for exponentiation.

2. Overview

In the subject/observer design pattern, an object, called the subject, maintains a list of interested parties, called observers, and notifies them automatically when a predefined condition, event or state change occurs. The subject provides a way for observers to register themselves with the subject. This pattern supports a clean separation between components, such as data storage and user interface.

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

Figure 1: Subject/Observer Design Pattern

The design pattern is realized in CoAP as follows:

Subject: In the context of CoAP, the subject is a resource located at some CoAP server. The state of the resource may change over time, ranging from infrequent updates to continuous state transformations.

Observer: The observer is a CoAP client that is interested in the current state of the resource at any given time.

Observation Relationship: A client registers itself with a resource by sending a modified GET request to the server. The request causes the server to establish an observation relationship between the client and the resource. The response to the GET request supplies the client with a representation of the current resource state.

Notification: Whenever the state of a resource changes, the server notifies each client that has an observation relationship to that resource. The notification is an additional response to the GET request; it supplies the client with a representation of the new resource state. The response echoes the token specified in the request, so the client can easily correlate notifications.

Figure 2 shows an example of a CoAP client establishing an observation relationship to a resource on a CoAP server and being notified, once upon registration and then whenever the state of the resource changes. The request to establish an observation relationship and all notifications are identified by the new Observe Option defined in this document.

```
   Client              Server
      |                  |
      | GET /temperature |
      |  Observe: 0      |  (establish observation relationship)
      |    Token: 0x4a   |
      +----------------->|
      |                  |
      |   2.05 Content   |
      |  Observe: 12     |  (initial notification of current state)
      |    Token: 0x4a   |
      |  Payload: 22.9 C |
      |<-----------------+
      |                  |
      |   2.05 Content   |
      |  Observe: 44     |  (notification upon state change)
      |    Token: 0x4a   |
      |  Payload: 22.8 C |
      |<-----------------+
      |                  |
      |   2.05 Content   |
      |  Observe: 60     |  (notification upon state change)
      |    Token: 0x4a   |
      |  Payload: 23.1 C |
      |<-----------------+
      |                  |
```

Figure 2: Observing a Resource in CoAP

3. Observation Relationships

3.1. Establishment

A client registers itself with a resource by performing a GET request that includes an Observe Option. (See Section 5 for the option definition.) When a server receives such a request, it services the request like a GET request without this option and, if the resulting response indicates success, establishes an observation relationship between the client and the target resource.

The token specified by the client in the GET request will be echoed by the server in the initial response and in all notifications sent to the client as part of the observation relationship. The server will also include an Observe Option in each response/notification to indicate that the observation relationship was successfully established. (See Section 4 for the details of notifications.)

A server that is unable or unwilling to establish an observation

relationship between a client and a resource MUST silently ignore the Observe Option and process the GET request as usual. The resulting response will not include an Observe Option, implying that no observation relationship was established.

3.2. Maintenance

A client MAY refresh an observation relationship at any time. (For example, when it didn't receive a notification for some time, it is not clear whether the resource never changed or the server rebooted and lost its state -- this is similar to the keep-alive problem of transport protocols, see e.g. the discussion in [RFC1122].) However, it is RECOMMENDED that the client does not refresh the relationship for the time specified in the Max-Age Option of the most recent notification received, including the initial response.

A client refreshes an observation relationship simply by repeating the GET request with the Observe Option. When a server receives such a repeated request (i.e. a GET request from a client for which an observation relationship already exists), it MUST NOT establish a second relationship but replace or update the existing one. If a GET request does not include an Observe Option, the server MUST end any relationship that may exist between the client and the target resource.

The exact rules for determining if two requests relate to the same observation relationship are as follows:

- The request URI of the two requests MUST match.

- The sources of the two requests MUST match. How this is determined depends on the security mode used (see Section 10 of [I-D.ietf-core-coap]): With NoSec, the IP address and port number of the request sources must match. With other security modes, in addition to the IP address and UDP port number matching, the requests must have the same security context.

- The Message IDs and any Token Options in the two requests MUST NOT be taken into account.

3.3. Termination

The observation relationship between a client and a resource ends when one of the following conditions occurs:

- The server sends a notification response with an error response code (4.xx or 5.xx).

- The client rejects a confirmable notification with a RST message.

- The last attempt of transmitting a confirmable notification to the client times out. (In this case, the server MAY also end all other observation relationships that the client has.)

A client MAY terminate an observation relationship by performing one of the following actions:

- The client rejects a confirmable notification with a RST message.

- The client performs a GET request on the resource without an Observe Option.

4. Notifications

When an observation relationship is established between a client and a resource, the client is notified of resource state changes by additional responses sent in reply to the GET request to the client. Each such notification response MUST include an Observe Option and echo the token specified by the client in the request. The order in which observers are notified about a state change 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 is changed in a way that would cause a basic GET request to return an error (for example, when the resource is deleted), the server SHOULD notify the client by sending a notification with an appropriate error code and MUST end the observation relationship.

The representation format (i.e. the media type) used in any notification resulting from an observation relationship MUST be the same format 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 5.00 (Internal Server Error) notification response and MUST end the observation relationship.

A notification can be sent confirmable or non-confirmable. A server can employ different strategies for notifying a client; see Section 4.1 below. The objective is that the state observed by the client eventually becomes consistent with the actual state of the resource.

If a client does not recognize the token in a confirmable notification, it MUST NOT acknowledge the message and SHOULD reject the message with a RST message (in which case the server MUST end the

observation). Otherwise, the client MUST acknowledge the message with an ACK message as usual. See Section 4.2 for details on the retransmission of confirmable messages.

Note that notifications may arrive in a different order than sent by the server due to network latency. If a notification arrives before the initial response to a request, the client can take the notification as initial response in place of the actual initial response. The client must be prepared to receive notifications after an error notification or after the client has requested the server to end the observation relationship. See Section 4.3 for further details on message reordering.

Notifications MAY be cached by CoAP end-points under the same conditions as with all responses. This is detailed in Section 4.4.

4.1. Strategies

The objective when notifying clients of state changes is that the state observed by the client eventually becomes consistent with the actual state of the resource. This allows the server some liberties in how it sends notifications, as long as it works towards this objective.

A notification can be sent confirmable or non-confirmable. 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 omit notifying a client of a state change if it knows that it will send another notification soon (e.g., when the state is changing frequently or maybe even continuously). Similarly, it MAY choose to notify a client about the same state change more than once. For example, when state changes occur in bursts, the server can omit some notifications, send others 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.

4.2. Retransmission

According to the core CoAP protocol, 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 changed in the meantime. There are many reasons why a client might not acknowledge a confirmable message, ranging from short interruptions in the network to a permanent failure of the client.

When a server is retransmitting a confirmable message with a notification, is waiting for an acknowledgement, and wants to notify the client of a state change using a new confirmable message, it MUST stop retransmitting the old notification and MUST attempt to transmit 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 observation relationship is ended.

4.3. Reordering

Messages with notifications can arrive in a different order than they were sent. Since the objective is eventual consistency, a client can safely discard a notification that arrives later than a newer notification.

For this purpose, the server keeps a single 16-bit unsigned integer variable. The variable is incremented approximately every second, wrapping around every 2^16 seconds (roughly 18.2 hours). The server MUST include the current value of the variable as the value of the Observe Option each time it sends a notification. The server MUST NOT send two notifications with the same value of the variable that pertain to the same client.

A client MAY discard a notification as outdated (not fresh) under the following condition:

```
(V1 - V2) % (2^16) < (2^15)    and    T2 < (T1 + (2^14))
```

where T1 is a client-local timestamp of the latest valid notification received for this resource (in seconds), T2 a client-local timestamp of the current notification, V1 the value of the Observe Option of the latest valid notification received, and V2 the value of the Observe Option of the current notification. The first condition essentially verifies that V2 > V1 holds in 16-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 (but is not needed any more, because reordering is not expected to occur on the order of 2^14 seconds). Note that the constants of 2^14 and 2^15 are non-critical, as is the even speed of

the clocks involved; e.g., the second check can be implemented by marking a response as fresh on reception and downgrading all responses periodically every, say, 2^13 seconds; once it has been downgraded twice, it no longer participates in freshness checks.

4.4. Caching

As notifications are just additional responses to a GET request, the same rules on caching apply as to all responses: CoAP end-points MAY cache the responses and thereby reduce the response time and network bandwidth consumption. Both the freshness model and the validation model are supported.

When a response is fresh in the cache, GET requests can be satisfied without contacting the origin server. This is particularly useful when the cache is located at an CoAP intermediary such as a proxy or reverse proxy. (Note that the freshness of the stored response is determined by its Max-Age Option, not the existence of an observation relationship. So a request can cause the end-point to refresh cache and observation relationship even while having an relationship.)

When an end-point has one or more responses stored, it can use the ETag Option to give the origin server an opportunity to select a stored response to be used. The end-point SHOULD add an ETag Option specifying the entity-tag of each stored response that is applicable. It MUST keep those responses in the cache until it terminates the observation relationship or sends a GET request with a new set of entity-tags. 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, it sends 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 request.

5. Observe Option

No.	C/E	Name	Format	Length	Default
10	Elective	Observe	uint	0-2 B	(none)

Table 1: New Options

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 once, but also lets the server notify the client of changes to the resource state.

In a response, the Observe Option indicates that an observation relationship has been established. The option's value is a sequence number that can be used for reordering detection (see Section 4.3). The value is encoded as a variable-length unsigned integer (see Appendix A of [I-D.ietf-core-coap]).

Since the Observe Option is elective, a GET request that includes the Observe Option will automatically fall back to a basic GET request if the server does not support observations.

6. Interactions with other CoAP features

6.1. Request Methods

If a client has an observation relationship with a resource and performs a POST, PUT or DELETE request on that resource, the request MUST NOT affect the observation relationship. However, since such a request can affect the observed resource, it can cause the server to send a notification with a resource state representation or end the observation relationship with an error notification (e.g., when a DELETE request is successful and an observed resource no longer exists).

Note that a client cannot perform a GET request on a resource to retrieve a representation of the current resource state without affecting an existing observation relationship to that resource: the client is already notified by the server with a fresh representation whenever the state changes. If the client wants to make sure that is has a fresh representation and wants to continue being notified, it should refresh the observation relationship (see Section 3.2). If the client wants to make sure it has a fresh representation and does not want to continue being notified, it should perform a GET request without an Observe Option (see Section 3.3).

6.2. Block-wise Transfers

Resources that are the subject of an observation relationship 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:

o As with basic GET transfers, the client can indicate its desired block size in a Block option in the GET request. If the server

supports block-wise transfers, it SHOULD take note of the block size not just for the initial response but also for further notifications in this observation relationship.

- Notification responses can make use of the Block option. The client SHOULD use the Observe option value from the last block. All blocks in a notification response SHOULD also carry an ETag option to ensure they are reassembled correctly.

6.3. Resource Discovery

Clients can discover resources that are interesting to observe using CoRE Resource Discovery [I-D.ietf-core-link-format]. Links with the "obs" attribute indicate resources that MUST support the mechanism in this document and are RECOMMENDED to change their state at least once in a while.

The "obs" attribute is used as a flag, and thus it has no value component. The attribute MUST NOT appear more than once in a link.

7. Security Considerations

The security considerations of the base protocol [I-D.ietf-core-coap] apply.

Note that the considerations about amplification attacks are somewhat amplified in an observation relationship. In NoSec mode, a server MUST therefore strictly limit the number of messages generated from an observation relationship that it sends between receiving packets that confirm the actual interest of the recipient 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 observation relationships. Servers MAY want to access-control this creation of state. As degraded behavior, the server can always fall back to a basic GET request (without an Observe option) if it is unwilling or unable to establish the observation relationship, including if resources for state 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.

8. IANA Considerations

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

Number	Name	Reference
10	Observe	[RFCXXXX]

Table 2: New CoAP Option Numbers

The following entry is added to the CoRE Link Format Attribute registry:

Name	Reference
obs	[RFCXXXX]

Table 3: New CoRE Link Format Attributes

9. Acknowledgements

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

Thanks to Daniele Alessandrelli, Peter Bigot, Angelo Castellani, Gilbert Clark, Esko Dijk, Brian Frank and Salvatore Loreto for helpful comments and discussions that have shaped the document.

Klaus Hartke was funded by the Klaus Tschira Foundation.

10. References

10.1. Normative References

[I-D.ietf-core-block]
Shelby, Z. and C. Bormann, "Blockwise transfers in CoAP", draft-ietf-core-block-02 (work in progress), March 2011.

[I-D.ietf-core-coap]
Shelby, Z., Hartke, K., Bormann, C., and B. Frank, "Constrained Application Protocol (CoAP)", draft-ietf-core-coap-05 (work in progress), March 2011.

[I-D.ietf-core-link-format]
Shelby, Z., "CoRE Link Format", draft-ietf-core-link-format-03 (work in progress), March 2011.

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

10.2. Informative References

[REST] Fielding, R., "Architectural Styles and the Design of Network-based Software Architectures", 2000, <http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm>.

[RFC1122] Braden, R., "Requirements for Internet Hosts - Communication Layers", STD 3, RFC 1122, October 1989.

[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.

Appendix A. Examples

```
   Client  Server
      |       |
      |       |
      +----->|     Header: GET (T=CON, Code=1, MID=0x1633)
      | GET  |       Token: 0x4a
      |       |         Uri: coap://sensor.example/temperature
      |       |    Observe: 0
      |       |
      |       |
      |<-----+     Header: 2.05 Content (T=ACK, Code=69, MID=0x1633)
      | 2.05 |       Token: 0x4a
      |       |    Observe: 27
      |       |    Payload: "22.9 C"
      |       |
      |       |
      |<-----+     Header: 2.05 Content (T=NON, Code=69, MID=0x7b50)
      | 2.05 |       Token: 0x4a
      |       |    Observe: 28
      |       |    Payload: "22.8 C"
      |       |
      |       |
      |<-----+     Header: 2.05 Content (T=NON, Code=69, MID=0x7b51)
      | 2.05 |       Token: 0x4a
      |       |    Observe: 29
      |       |    Payload: "22.5 C"
      |       |
```

Figure 3: Simple observation with non-confirmable notifications

A.1. Proxying

```
Client  Proxy  Server
   |      |      |
   |      |      |
   |      +----->|     Header: GET (T=CON, Code=1, MID=0x5fb8)
   |      | GET  |      Token: 0x1a
   |      |      |        Uri: coap://sensor.example/status
   |      |      |    Observe: 0
   |      |      |
   |      |      |
   |      |<-----+     Header: 2.05 Content (T=ACK, Code=69, MID=0x5fb8)
   |      | 2.05 |      Token: 0x1a
   |      |      |    Observe: 42
   |      |      |    Max-Age: 120 sec
   |      |      |    Payload: "ready"
   |      |      |
   |      |      |
   +----->|      |     Header: GET (T=CON, Code=1, MID=0x1633)
   | GET  |      |      Token: 0x9a
   |      |      |  Proxy-Uri: coap://sensor.example/status
   |      |      |
   |      |      |
   |<-----+      |     Header: 2.05 Content (T=ACK, Code=69, MID=0x1633)
   | 2.05 |      |      Token: 0x9a
   |      |      |    Max-Age: 113 sec
   |      |      |    Payload: "ready"
   |      |      |
   |      |      |
   |      |<-----+     Header: 2.05 Content (T=NON, Code=69, MID=0x5fc0)
   |      | 2.05 |      Token: 0x1a
   |      |      |    Observe: 1780
   |      |      |    Max-Age: 120 sec
   |      |      |    Payload: "busy"
   |      |      |
   |      |      |
   +----->|      |     Header: GET (T=CON, Code=1, MID=0x1634)
   | GET  |      |      Token: 0x9b
   |      |      |  Proxy-Uri: coap://sensor.example/status
   |      |      |
   |      |      |
   |<-----+      |     Header: 2.05 Content (T=ACK, Code=69, MID=0x1634)
   | 2.05 |      |      Token: 0x9b
   |      |      |    Max-Age: 89 sec
   |      |      |    Payload: "busy"
   |      |      |
```

Figure 4: A proxy observes a resource to keep its cache up to date

```
Client  Proxy  Server
   |      |      |
   |      |      |
   +----->|      |     Header: GET (T=CON, Code=1, MID=0x1633)
   | GET  |      |      Token: 0x6a
   |      |      |  Proxy-Uri: coap://sensor.example/status
   |      |      |    Observe: 0
   |      |      |
   |      |      |
   |<- - -+      |     Header: (T=ACK, Code=0, MID=0x1633)
   |      |      |
   |      |      |
   |      +----->|     Header: GET (T=CON, Code=1, MID=0xaf90)
   |      | GET  |      Token: 0xaa
   |      |      |        Uri: coap://sensor.example/status
   |      |      |    Observe: 0
   |      |      |
   |      |      |
   |      |<-----+     Header: 2.05 Content (T=ACK, Code=69, MID=0xaf90)
   |      | 2.05 |      Token: 0xaa
   |      |      |    Observe: 67
   |      |      |    Payload: "ready"
   |      |      |
   |      |      |
   |<-----+      |     Header: 2.05 Content (T=CON, Code=69, MID=0xaf94)
   | 2.05 |      |      Token: 0x6a
   |      |      |    Observe: 346
   |      |      |    Payload: "ready"
   |      |      |
   |      |      |
   +- - ->|      |     Header: (T=ACK, Code=0, MID=0xaf94)
   |      |      |
   |      |      |
   |      |<-----+     Header: 2.05 Content (T=CON, Code=69, MID=0x5a20)
   |      | 2.05 |      Token: 0xaa
   |      |      |    Observe: 1460
   |      |      |    Payload: "busy"
   |      |      |
   |      |      |
   |      +- - ->|     Header: (T=ACK, Code=0, MID=0x5a20)
   |      |      |
   |      |      |
   |<-----+      |     Header: 2.05 Content (T=CON, Code=69, MID=0xaf9b)
   | 2.05 |      |      Token: 0x6a
   |      |      |    Observe: 2011
   |      |      |    Payload: "busy"
   |      |      |
   |      |      |
```

```
   +- - ->|      |     Header: (T=ACK, Code=0, MID=0xaf9b)
   |      |      |
```

Figure 5: A client observes a resource through a proxy

Appendix B. Changelog

Changes from ietf-01 to ietf-02:

- Removed the requirement of periodic refreshing (#126).
- The new "Observe" Option replaces the "Lifetime" Option.
- 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).

Authors' Addresses

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

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

Zach Shelby
Sensinode
Kidekuja 2
Vuokatti 88600
Finland

Phone: +358407796297
Email: zach@sensinode.com