CoRE Working Group	K. Hartke
Internet-Draft	Universitaet Bremen TZI
Intended status: Standards Track	Z. Shelby
Expires: August 11, 2011	Sensinode
	February 07, 2011

Observing Resources in CoAP
draft-ietf-core-observe-01

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].

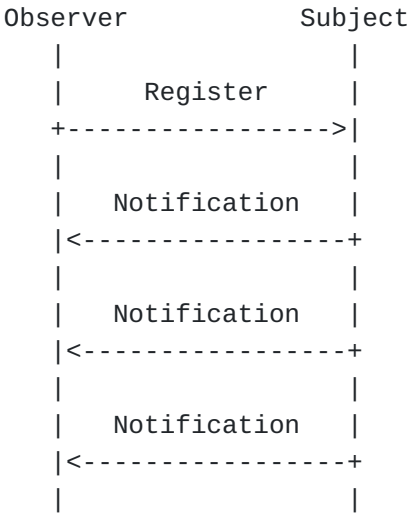
This short specification describes an architecture and a protocol design that realizes the well-known subject/observer design pattern within the REST-based environment of CoAP.

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].

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 pattern supports a clean separation between components, such as data storage and user interface.

The subject typically provides a method for observers to register themselves with the subject (see Figure 1). The order in which observers receive notifications is not defined; the subject is free to use any method to determine the order.



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 changes to continuous state updates.

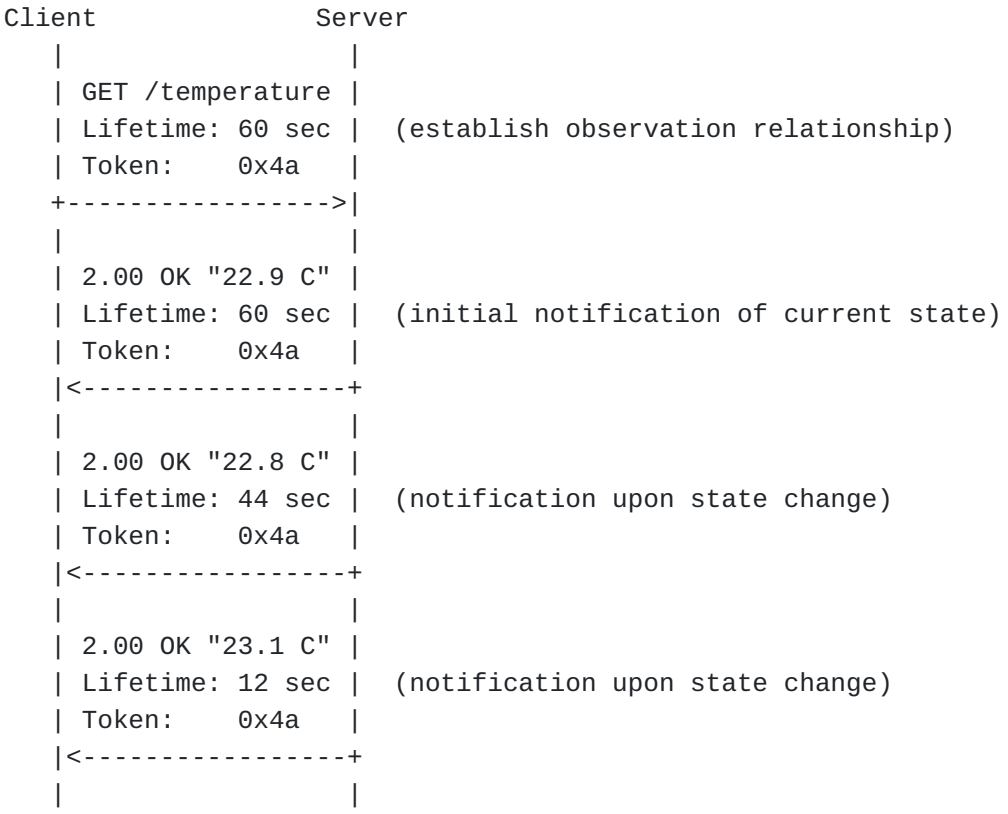
Observer: The observer is a CoAP client that is interested in the current state of the resource at any given time.

Observation Relationship: A client registers itself with a resource by sending a modified GET request to the server hosting the resource. 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 the 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 by the client in the request, so the client can easily correlate notifications.

Lifetime: For robustness, an observation relationship is automatically ended after a negotiated duration of time. A client needs to refresh the relationship before the lifetime ends if it wants to be kept in the list of observers. The server includes the remaining lifetime duration in each notification.

Figure 2 shows an example of a CoAP client establishing an observation relationship with a resource on a CoAP server and then being notified, once upon registration and then whenever the state of the resource changes.



3. Observation Relationships

3.1. Establishment

A client registers itself with a resource by sending a GET request that includes a Lifetime Option. (See Section 5 for the option definition.) When a server receives such a request, it satisfies the request as with a basic GET request and, upon success, establishes an observation relationship between the client and the target resource.

The Lifetime Option indicates the duration for which the server is requested to maintain the observation relationship before it is ended. The server MUST NOT establish the relationship with a duration longer than requested, although it MAY choose to cut short the remaining lifetime upon registration (or any time while the relationship is established). The server MUST include the remaining lifetime in each response sent in reply to the GET request, including the initial response.

A server that is unable or unwilling to establish an observation relationship between a client to a resource MUST silently ignore the Lifetime Option and process the GET request as usual. The resulting response will not include a Lifetime Option, implying that no observation relationship was established.

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 during the lifetime of the observation relationship. See Section 4 for the details on notifications.

3.2. Maintenance

For robustness, an observation relationship has to be maintained through periodic refreshing. If the relationship is not refreshed, it ends after the duration that is negotiated using the Lifetime Option. A client refreshes an observation relationship by repeating the original GET request shortly before the observation lifetime ends.
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 with the new duration.
The exact rules for determining if two requests request to establish 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.

A client MAY refresh an observation relationship at any time before the lifetime ends, for example, when it didn't receive a notification for some time. 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.

3.3. Termination

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

- The age of the observation relationship becomes greater than the negotiated lifetime.

- The server sends a notification response with a non-success 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.

A client MAY terminate an observation relationship before its lifetime ends. It can do so by performing one of the following actions:

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

*The client refreshes the observation relationship with a value of 0 seconds, which will cause the server to end any observation relationship immediately after returning the initial response.

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 echo the token specified in the request and specify the remaining duration for which the server maintains the observation relationship. 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 2.00 (OK) or 2.03 (Valid) response code. However, in the event the state of a resource is changed in a way that would cause a basic GET request to return an error code (for example, when the resource is deleted), the server SHOULD notify the client with 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 response during the lifetime of an observation relation MUST be the same format used in the initial response to the GET request. If the server is unable to continue sending notifications in the same representation format, it SHOULD send a 5.00 (Internal Server Error) notification response and MUST end the observation relationship.

A notification may be confirmable or non-confirmable, and the server can employ different strategies in how it notifies 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 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 normal. 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. A client must be prepared to receive notifications before the initial response to a GET request, 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. 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 the objective.
A notification may 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 the 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.

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, 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 remaining lifetime indicated by

notifications as a result of an observation request MUST be strictly decreasing and the client SHOULD specify a token in a refresh request that is different from the token in the previous request.
A client MAY discard a notification under the following conditions:

* The client receives a notification with a token other than that specified in the most recent request.

* The client receives a notification with the right token but also with an indicated remaining lifetime duration longer than the duration specified in the previous notification.

4.4. Caching

As notifications are just additional responses to a GET request, the same rules on caching apply as to basic 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. The observation mechanism ensures that the cache has a fresh response for most of the duration of the observation lifetime. 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 the observation lifetime ends, the relationship is terminated or a refresh request with a new set of entity-tags is. When the observed resource changes its state and the origin server is about to send a 2.00 (OK) 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 observation request.

5. Lifetime Option

No.	C/E	Name	Format	Length	Default
10	Elective	Lifetime	uint	1-4 B	0

Options

The Lifetime 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 for the duration specified in the option.
In a response, the Lifetime Option indicates a lower bound (e.g., by rounding down) for the remaining observation lifetime. (Note that the server can always choose to cut short the observation lifetime before it echoes this lifetime back in a response.)
The option's value is encoded as a variable-length unsigned integer (see Appendix A of [I-D.ietf-core-coap]) that indicates a duration of time measured in seconds.
Note that a Lifetime value of 0 is indicated by leaving out the Lifetime option, which then defaults to 0.
Since the Lifetime Option is elective, a GET request that includes the Lifetime 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 the observed resource no longer exists).
Note that a client cannot perform a GET request on a resource to get a representation of the current resource state without affecting the lifetime of an existing observation relation 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 as described in 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 with a lifetime duration of 0 seconds as described in 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:

*As with basic GET transfers, the client can indicate its desired block size in a Block option in the GET request. If it implements Block, the server 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. With increasing block numbers, the Lifetime option value MUST stay the same or decrease. The client SHOULD use the Lifetime information 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 interesting resources 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 (no Lifetime 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	Lifetime	[RFCXXXX]

New CoAP Option Numbers

The following entry is added to the CoRE Target Attribute registry:

Name	Reference
obs	[RFCXXXX]

New CoRE Target Attributes

9. Acknowledgements

Carsten Bormann was an original author of this draft and is acknowledged for significant contribution to this document.
Klaus Hartke was funded by the Klaus Tschira Foundation.

10. References

10.1. Normative References

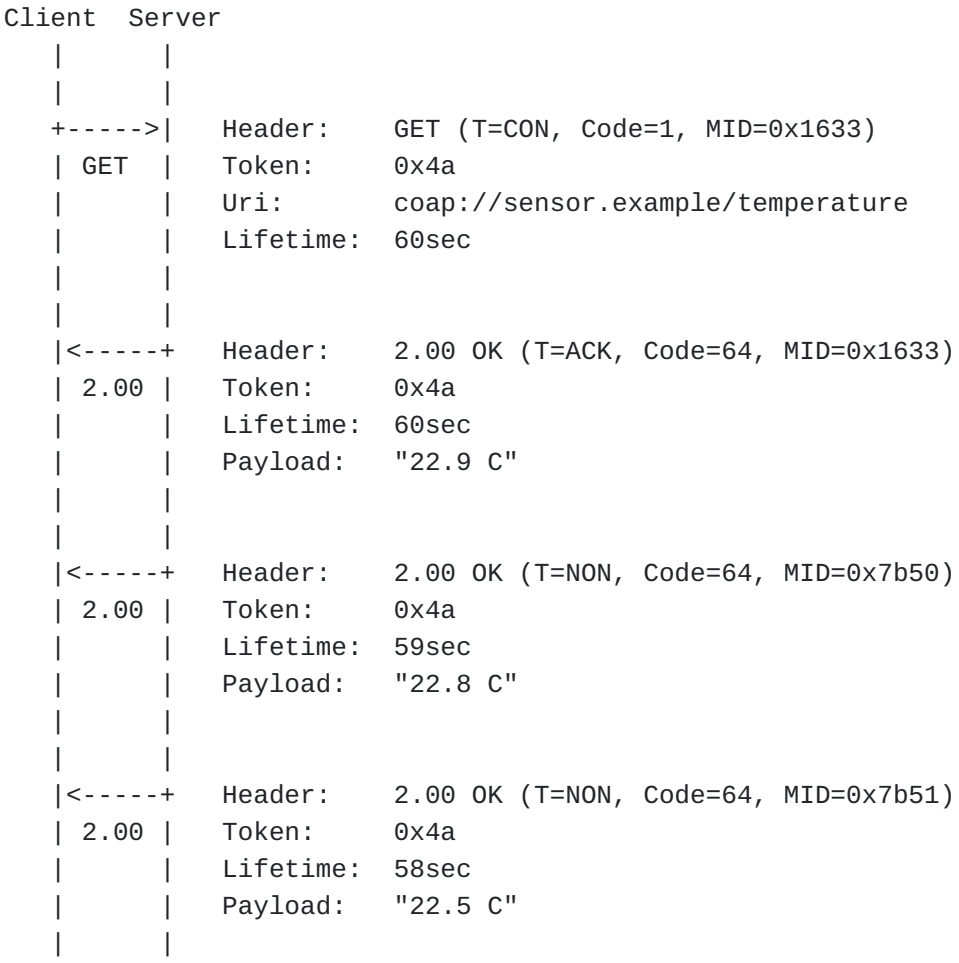
[RFC2119]	Bradner, S., "Key words for use in RFCs to Indicate Requirement Levels", BCP 14, RFC 2119, March 1997.
[I-D.ietf-core-coap]	Shelby, Z, Hartke, K, Bormann, C and B Frank, "Constrained Application Protocol (CoAP)", Internet-Draft draft-ietf-core-coap-04, January 2011.
[I-D.ietf-core-link-format]	Shelby, Z, "CoRE Link Format", Internet-Draft draft-ietf-core-link-format-02, December 2010.
[I-D.ietf-core-block]	Shelby, Z and C Bormann, "Blockwise transfers in CoAP", Internet-Draft draft-ietf-core-block-01, January 2011.

10.2. Informative References

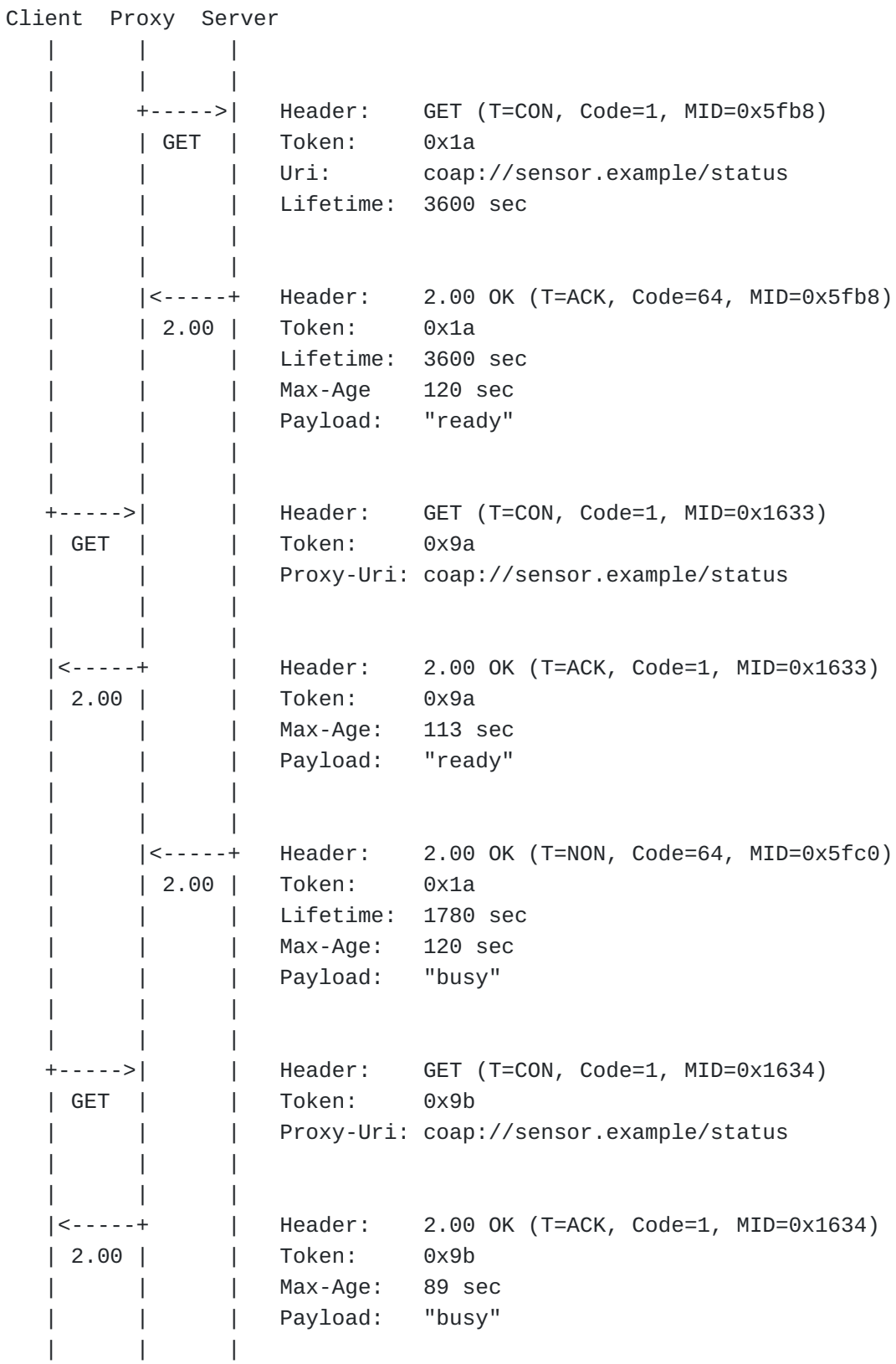
[RFC2616]	Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P. and T. Berners-Lee, "Hypertext Transfer Protocol -- HTTP/1.1", RFC 2616, June 1999.
[RFC5989]	Roach, A.B., "A SIP Event Package for Subscribing to Changes to an HTTP Resource", RFC 5989, October 2010.
[REST]	Fielding, R, "Architectural Styles and the Design of Network-based Software Architectures", 2000.

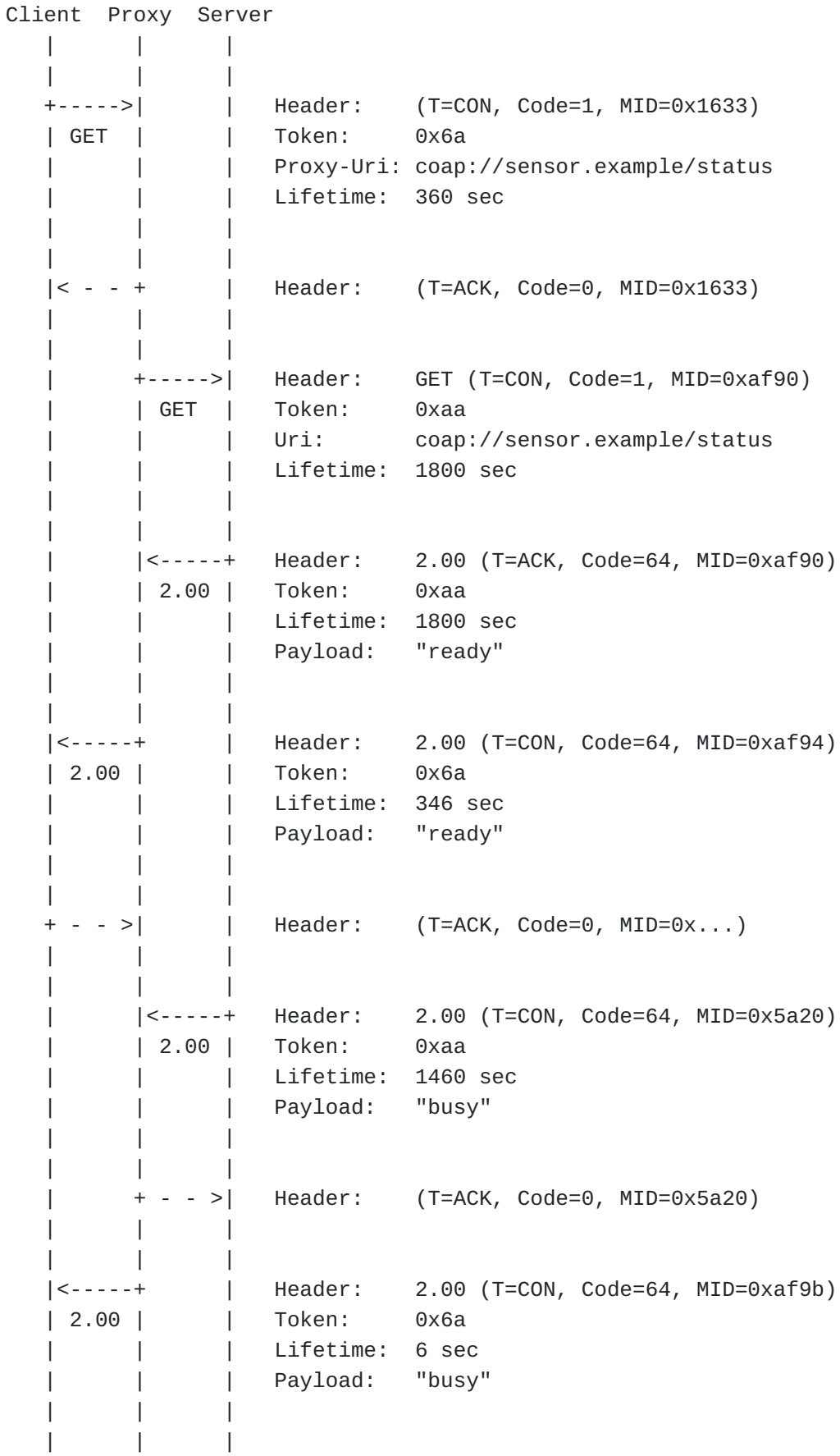
	(Seminal dissertation introducing the REST architectural style.)

Appendix A. Examples



Appendix A.1. Proxying


Client Proxy Server
GET
Header: GET (T=CON, Code=1, MID=0x5fb8)
Token: 0x1a
Uri: coap://sensor.example/status
Lifetime: 3600 sec
2.00
Header: 2.00 OK (T=ACK, Code=64, MID=0x5fb8)
Token: 0x1a
Lifetime: 3600 sec
Max-Age 120 sec
Payload: "ready"
GET
Header: GET (T=CON, Code=1, MID=0x1633)
Token: 0x9a
Proxy-Uri: coap://sensor.example/status
2.00
Header: 2.00 OK (T=ACK, Code=1, MID=0x1633)
Token: 0x9a
Max-Age: 113 sec
Payload: "ready"
2.00
Header: 2.00 OK (T=NON, Code=64, MID=0x5fc0)
Token: 0x1a
Lifetime: 1780 sec
Max-Age: 120 sec
Payload: "busy"
GET
Header: GET (T=CON, Code=1, MID=0x1634)
Token: 0x9b
Proxy-Uri: coap://sensor.example/status
2.00
Header: 2.00 OK (T=ACK, Code=1, MID=0x1634)
Token: 0x9b
Max-Age: 89 sec
Payload: "busy"


Client Proxy Server
GET
Header: (T=CON, Code=1, MID=0x1633)
Token: 0x6a
Proxy-Uri: coap://sensor.example/status
Lifetime: 360 sec
Header: (T=ACK, Code=0, MID=0x1633)
GET
Header: GET (T=CON, Code=1, MID=0xaf90)
Token: 0xaa
Uri: coap://sensor.example/status
Lifetime: 1800 sec
2.00
Header: 2.00 (T=ACK, Code=64, MID=0xaf90)
Token: 0xaa
Lifetime: 1800 sec
Payload: "ready"
2.00
Header: 2.00 (T=CON, Code=64, MID=0xaf94)
Token: 0x6a
Lifetime: 346 sec
Payload: "ready"
Header: (T=ACK, Code=0, MID=0x...)
2.00
Header: 2.00 (T=CON, Code=64, MID=0x5a20)
Token: 0xaa
Lifetime: 1460 sec
Payload: "busy"
Header: (T=ACK, Code=0, MID=0x5a20)
2.00
Header: 2.00 (T=CON, Code=64, MID=0xaf9b)
Token: 0x6a
Lifetime: 6 sec
Payload: "busy"

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

Appendix B. Changelog

Changes from ietf-00 to ietf-01:

*Name of the option is now simply "Lifetime".

*Terminology changed from "subscriptions" to "observation relationships" (#33).

*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 Hartke Universitaet Bremen TZI Postfach 330440
Bremen, D-28359 Germany Phone: +49-421-218-63905 EMail:
hartke@tzi.org

Zach Shelby Shelby Sensinode Kidekuja 2 Vuokatti, 88600 Finland
Phone: +358407796297 EMail: zach@sensinode.com