COAP-OBSERVE FEATURE IMPLEMENTATION ON THINKIP SOFTWARE

RELATORE: Ch.mo Prof. Michele Zorzi
LAUREANDO: Felix Mendoza

Telecommunication Engineering (Laurea Magistrale)

A.A. 2012-2013

UNIVERSITÀ DEGLI STUDI DI PADOVA

Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria delle Telecomunicazioni

TESI DI LAUREA

COAP-OBSERVE FEATURE IMPLEMENTATION ON THINKIP SOFTWARE

RELATORE: Prof. Michele Zorzi

LAUREANDO: Felix Mendoza

A.A. 2012-2013

To my family

Contents

Abstract

Patavina Technologies ([3]) is a software house that designs telecomunication systems. It is a spinoff of the University of Padua since both founders (*Lorenzo Vangelista* and *Michele Zorzi*) have been teaching in the institute courses related to the telecomunications and computer science fields for several years.

One of its current projects (*ThinkIP*) focuses on *INTERNET OF THINGS* and *6LOWPAN* protocols. The project is presented as a software package defined by the following components:

- Resource Access Module
- OS Abstraction Layer
- Debug Interface
- Hardware Abstraction Layer
- Protocol Stack
  - IPv6
  - RPL (Routing Protocol over Lossy and Low Power networks)
  - TCP and UDP
  - Security Mechanisms
  - IPv4 e DHCP
  - Interfaces for USB, Ethernet and UART for UMTS modules
  - Proxy HTTP/CoAP transparency
  - CoAP/HTTP gateway
  - Power Management

CoAP (*Constrained Application Protocol*) is a specialized web transfer protocol for constrained nodes and networks (e.g. low-power, lossy). The nodes often have a 8-bit microcontroller with small amounts of ROM and RAM, while constrained networks such as 6LoWPAN often have a high packet error rate and a typical throuhput of 10s of kbits/s. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation. CoAP provides a request/response interaction model between application endpoints , supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead and simplicity for constrained enviroments.

The objective of this thesis is implementing *Observe* on the software package of *Patavina Technologies. Observe* is a simple protocol extension that enables CoAP clients to "keep an eye on" a resource following a best-effort approach.

Chapter 1

Introduction

The main focus of this thesis will be the CoAP component of ThinkIP, specifically its *Observe* feature implementation into it.

CoAP is a fairly new protocol under development and it should be considered as a work in progress. The latest updates comes from Internet-Drafts (Internet-Draft are working documents, mainly from the IETF). Since the protocol is frequently updated, Patavina Technologies has been working with CoAP-08 (version 08). CoAP is intended to provide *RESTful* services ([9])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. A more detailed picture of the CoAP protocol can be found in *Chapter two.*

The basic flow of communication in the CoAP protocol is given by: a resource request sent from a client to a server, which will return a response with the respresentation of the resource requested (if possible). However, this model does not suit well the scenario in which a client needs the current representation of the resource in a period of time, these continuous polling would significantly generate complexity and overhead, scenario that must be avoided in a constrained network.

The observe feature in the CoAP protocol is a fair alternative to continuous polling, once a client makes a request, the server stores the client ID and repeatedly sends responses to the interested client without the need of further requests. We should mention that the thesis focuses on the version 03 of the observe feature, since it is the most time accurate version regarding CoAP-08. The feature also takes into account the scenario of packet loss or client/service malfunction. There are different ways to treat this case

- Ack-less notifications: One solution would be to repeatedly send responses without the need for the acknowlegde of the notifications. Since it is possible for a client to malfunction and go offline (a client could also just lose interest for a resource), a timer must be established and everytime it expires the server would require a response from the client to confirm its interest.
- Acked Notifications: The server expects an acknowledgement for every notification it sends.

It is not simple to determine which of the solutions above is better, at a first sight the *Ack-less* option would seem to be better since it has less overhead due to the lack of acknowledgements, but once a client goes offline or lose interests in the resource, the server would continue to send notification until the timer expires. This problem can be fixed by choosing the right timer, the latter though depends on the network and the network nodes.

It is also possible for a server malfunction to happen, so each client has a timer that is reset each time the client receives a notification. If the timer expires the client may assume that the server has gone offline. The value of such timer depends on the Max-OFE option of the observe feature. However, a more detailed picture of Observe-03 can be found in *Chapter three*.

The main focus of this thesis is a practical implementation of the observe feature into an existing and functioning software (that obviously implements the CoAP protocol). To achieve our objective we have worked with the ThinkIP software (developed by Patavina Technologies).

Patavina Technologies is an italian software house that recently focuses on the development of telecommunication software and systems, specially on new technologies (new standards, future standards or promising technologies) such as constrained networks. Its main product is *ThinkIP*, a software that relies on well known protocols to enable the communication between devices and create ad-hoc networks.

It is a portable software package (available to clients via licences) based on the *Internet of Things* concept and the *6LoWPAN* protocol. Furthermore, the work projectss of Patavina Technologies are very client-specific, since every client has different needs such as hardware and system design, hardware manufacturing, system integration, testing and forniture contact network. The software package is composed by the following:

- Resource Access Module

- OS Abstraction Layer (for Patavina Technology OS support)
- Debug Interface
- Hardware Abstraction Layer
- Protocol Stack
- Power Management

And its *Protocol Stack* includes:

- IPv6 (with ICMPv6)
- RPL (Routing Protocol ovr Lossy and Low Power Networks)
- TCP and UDP
- Security Mechanisms
- IPv4 and DHCP
- USB, Ethernet and UART interfaces for UMTS modules
- Proxy HTTP/CoAP (transparent)
- CoAP/HTTP gateway

The application field for the software package is limitless since it covers all applications linked to *Internet of things* such as home automation, monitoring (temperature, humidity, ecc), actuators (light, air conditioning systems), access control, metering. *ThinkIP* is written in C++ and has passed succesfully the 2012 ETSI *Plug Test* event for the CoAP protocol.

Patavina Technologies adopted the CoAP protocol into its software package to handle constrained nodes scenarios (further information about Patavina Technologies ThinkIP can be found in *Chapter four*), we will focus on the observe feature implementation of its CoAP component (*Chapter five*). The feature was added as part of the application layer but yet external to the existing layers inside of it, mainly because we wanted to avoid doing major changes to the existing code and reduce the complexity in future updates/modifications (the CoAP protocol and observe feature are being frequently updated).

Chapter 2

CoAP (version 08)

CoAP ([6]) is a software protocol that allows resource-constrained internet devices to interact. It is particularly targeted for WSN (*Wireless Sensor Network*) nodes. The protocol is designed to translate the data of such devices to *HTTP* for simplified integration with the web, while also meeting specialized requirements such as multicast support, very low overhead, and simplicity. These specialized requirements greatly improves the perfomance of embedded devices.

CoAP has the following features:

- Constrained web protocol fulfilling M2M (machine-to-machine) requirement.
- UDP binding with optional reliability supporting unicast and multicast requests.
- Asynchronous messages exchanges.
- Low header overhead and parsing complexity
- URI and Content-type support.
- Simple proxy and caching capabilities.
- A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.
- Security binding to Datagram Transport Layer Security (DTLS).

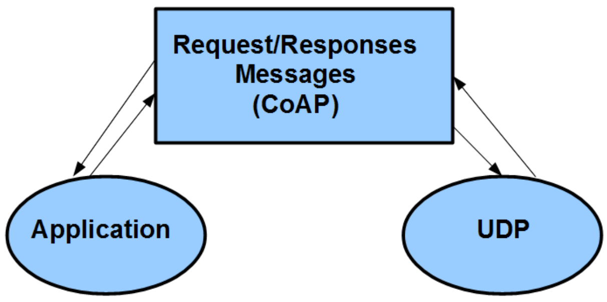


Figure 2.1: CoAP two-layer

2.1 Constrained Application Protocol

The interaction model of CoAP is similar to the client /server model of HTTP. However, M2M interactions typically result in a CoAP implementation acting in both client and server (end-points) roles. A CoAP request is equivalent to that of HTTP, and is sent by a client to request an action (using a method code) on a resource (identified by a URI) on a server. The server then sends a response with a response code that may include a resource rapresentation.

Unlike HTTP, CoAP deals with these interchanges asynchronously over a datagram-oriented transport such as UDP. This is done logically using a layer of messages that supports optional reliability (with exponential back-off).

CoAP defines four messages types:

- Confirmable
- Non-Confirmable
- Acknowledgement
- Reset

The exchange of these messages types are transparent to the request/response interactions. CoAP works between the application and UDP with a two-layer (Messages and Resquests/Responses) approach and is able to handle the asynchronous nature of the interactions (see Figure 2.1). Even though the messaging and request/response is just a component of the CoAP header.

2.1.1 Messaging Model

The CoAP messaging model is based on the exchange of messages over UDP between end-points. It uses a short fixed-length binary header (four bytes) that may be followed by compact binary options and a payload. Such message format is shared by the requests and the responses. Each message contains a Message ID (*MID*) used to detect duplicates and optional reliability.

Reliability is provided by marking a message as Confirmable (*CON*). A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an Acknowledgement message (*ACK*) with the same Message ID from the corresponding end-point. When a recipient is not able to process a Confirmable message, it replies with a Reset (*RST*) instead of an ACK. Such reliability is not necessary when dealing with Non-Confirmable (*NON*) messages, since these messages will not be needing an ACK. Even though NON messages are not acknowledged, they still have a message ID for duplicate detection.

As CoAP is based on UDP, it supports multicasts, enabling multicast CoAP requests.

2.1.2 Request/Response Model

CoAP request and response semantics are carried in CoAP messages, which include either a method code or response code, respectively. Optional (or default) request and response information, such as the URI and payload content-type are carried as CoAP options. A Token Option is used to match responses to requests independently from the underlying messages.

A request can be either a CON or NON message, and if immediately available, the response to a request can be delivered in the resulting ACK message (piggy-backed response).

If a server is not able to reply immediately to a request carried in a CON message, it simply responds with an empty ACK message so that the client can stop retransmitting the request. When the response is ready, the server sends it in a new CON message (an ACK will be needed to such new message). This is called a separate response.

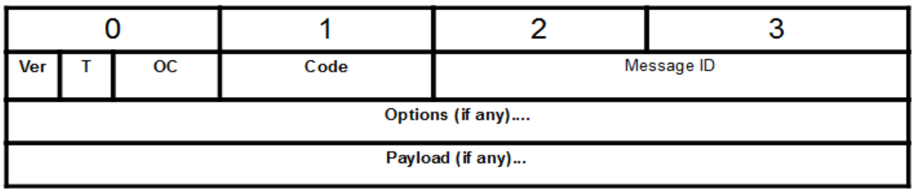


Figure 2.2: Message Format

2.1.3 Intermediaries and Caching

The protocol supports the caching of responses in order to efficiently fulfill requests. Simple caching is enabled using freshness and validity information carried with CoAP responses. A cache could be located in an end-point or an intermediary.

Proxying is useful in constrained networks for several reasons, including network traffic limiting, to improve performance, to access resources of sleeping devices or for security reasons. The proxying of requests on behalf of another CoAP end-point is supported in the protocol. The URI of the resource is included in the request, while the destination IP address is set to the proxy.

2.2 Message Syntax

CoAP messages are encoded in a simple binary format. A message consists of a fixed-size CoAP Header followed by options in Type-Length-Value (TLV) format and a payload. The number of options is determined by the header. The payload is determined by the bytes after the options (if any) and its length is calculated from the datagram length.

The fields in the header are defined as follows:

- Version (Ver): 2-bit unsigned integer. Indicates the CoAP version number. Implementations of this specification must set this field to 1. Other values are reserved for future versions.

- Type (T): 2-bit unsigned integer. Indicates the nature of the message type. CON (0), NON (1), ACK (2) or RST(3).

- Option Count (OC): 4-bit unsigned integer. Indicates the number of options after the header. If set to 0, there are no options and the payload (if any) immediately follows the header.

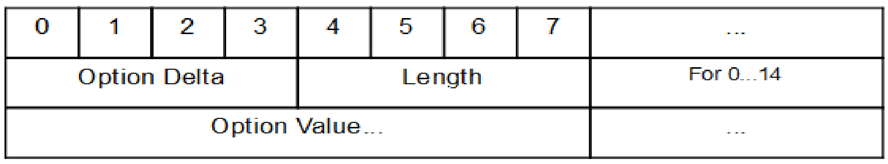


Figure 2.3: Message Option Format

- Code: 8-bit unsigned integer. Indicates if the message carries a request (1-31) or a response (64-191) or is empty (0). All other values are reserved.
- Message ID: 16-bit unsigned integer. Used for the detection of message duplication and for reliability.

While specific link layers make it beneficial to keep CoAP messages small enough to fit into their link layer packets, this is a matter of implementation quality. The CoAP specification itself provides only an upper bound to the message size. Messages larger than an IP fragment result in undesired packet fragmentation. A CoAP message, appropriately encapsulated, should fit within a single IP packet (avoiding IP fragmentation) and must fit within a single IP datagram. If the Path MTU is not known for a destination, an IP MTU of 1280 bytes should be assumed; if nothing is known about the size of the headers, then good upper bounds are 1152 btes for the mesage size and 1024 bytes for the payload size.

2.2.1 Option Format

Options must appear in order of their Option Number. A delta encoding is used between options, with the Option Number for each Option calculated as the sum of its Option Delta field and the Option Number of the preceding Option in the message, if any, or zero otherwise. Multiple options with the same Option Number can be included by using an Option Delta of zero. Following the Option Delta, each option has a Length field which specifies the length of the Option Value, in bytes. The length field can be extended by one byte for values longer than 14 bytes. The Option Value immediately follows the Length field (see figure 2.3.

The fields in an option are defines as follows:

- Option Delta: 4-bit unsigned integer. Indicates the difference between the Option Number of this option and the previous option (or zero for the first option). In other words, the Option Number is calculated by simply summing the Option Delta fields of this and previous options before it. The Option Numbers 14, 28, 42, ... are reserved for no-op options when they are sent with an empty value (they are ignored) and can be used as "fenceposts" if deltas larger than 15 would otherwise be required.

- Length: Indicates the length of the Option Value, in bytes. Normally, Length is a 4-bit unsigned integer allowing value lengths of 0-14 bytes. When the Length field is set to 15, another byte is added as an 8-bit unsigned integer whose value is added to the 15, allowing option value lengths of 15-270 bytes.

2.3 Message Semantics

CoAP messages are exchanged asynchronously between CoAP end-points. They are used to transport CoAP requests and responses. As CoAP is bound to non-reliable transports such as UDP, CoAP messages may arrive out of order, appear duplicated, or go missing without notice. For this reason, CoAP implements a lightweight reliability mechanism, without trying to re-create the full feature set of a transport like TCP. It has the following features:

- Simple stop-and-wait retransmission reliability with exponential back-off for CON messages.
- Duplicate detection for both CON and NON messages.
- Multicast support.

2.3.1 Reliable Messages

The reliable transmission of a message is initiated by marking the message as CON in the CoAP header. A recipient must acknowledge such a message with an ACK message (or a RST message if the recipient lacks context to process the message). The sender retransmits the CON message at exponentially increasing intervals, until it receives an ACK or RST message, or runs out of attempts.

Retransmission is controlled by two things that a CoAP end-point must keep track of for each CON message it sends while waiting for an ACK (or RST): a timeout and a retrasmission counter. For a new CON message, the initial timeout is set to a random number between *RESPONSE-TIMEOUT* and *RESPONSE_ TIMEOUT*∗*RESPONSE_ RANDOM_ FACTOR*, and the retrasmission counter is set to 0. When the timeout is triggered and the retrasmission counter is less than *MAX_ RETRANSMIT*, the message is retransmitted, the retransmission counter is incremented, and the timeout is doubled. If the retransmission counter reaches *MAX_ RETRANSMIT* on a timeout, or if the end-point receives a RST message, then the attempt to transmit the message is canceled and the application process a failure. On the other hand, if the end-point receives and ACK message in time, transmission is considered successful.

An ACK or RST message is related to a CON message by means of a Message ID along with additional address information of the corresponding end-point. The Message ID is a 16-bit unsigned integer that is generated by the sender of a CON message and included in the CoAP header. The Message ID must be echoed in the ACK or RST message by the recipient.

Several implementation strategies can be employed for generating Messages IDs. In the simplest case a CoAP end-point generates Message IDs by keeping a single Message ID variable, which is changed each time a new CON message is sent regardless of the destination address or port. End-points dealing with large numbers of transactions could keep multiple Message ID variables, for example per prefix or destination address. The same Message ID must not be re-used within the potential retransmission window:$RESPONSE\_TIMEOUT * RESPONSE\_RANDOM\_FACTOR * (2^{MAX\_RETRANSMIT} - 1) +$ the expected maximum round trip time.

A recipient must be prepared to receive the same CON message multiple times (the ACK message could have gone missing or did not reach the sender before the timeout). The recipient should acknowledge each duplicate copy of a CON message using the same ACK or RST message, but should process any request or response in the message only once. This rule may be relaxed in case the CON message transports a request that is idempotent.

2.3.2 Unreliable Messages

As a more lightweight alternative, a message can be transmitted less reliably by marking the message as NON. A NON message must not be acknowledged by the recipient. If a recipient lacks context to process the message properly, it may reject the message with a RST message or otherwise must silently ignore it.

There is no way to detect if a NON message was received or not at the CoAP-level. A sender may choose to transmit a NON message multiple times which, for this purpose, specifies a Message ID as well. The same rules for generating the Message ID apply.

A recipient must be prepared to receive the same NON message multiple times. As a general rule that may be relaxed based on the specific semantics of a message, the recipient should silently ignore any duplicated NON message, and should process any request or response in the message only once.

2.3.3 Messages Types

Separate from the message type, a message may carry a request, a response, or be empty. This is signaled by the Code field in the CoAP header and is relevant to the request/response model.

An empty message has the Code field set to 0. The OC field should be set to 0 and no bytes should be present after the Message ID field. The OC field and any bytes trailing the header must be ignored by any recipient.

- Confirmable (CON): Message that requires acknowledgement. When no packets are lost, each CON message elicits exactly one return message of type ACK or RST.

- Non-confirmable (NON): Message that does not require acknowledgement. This is particularly true for messages that are repeated regularly for sensor application requirements, such as repeated readings from a sensor where eventual arrival is sufficient. A NON message always carries either a request or response, and must not be empty.

- Acknowledgement (ACK): An ACK message acknowledges that a specific CON message (identified by its Message ID) arrived. It does not indicate success or failure of any encapsulated request. The ACK message must echo the Message ID of the CON message, and must carry a response or be empty.

- Reset (RST): A RST message indicates that a specific CON message was received, but some context is missing to properly process it. This condition is usually caused when the receiving node has rebooted and has forgotten some state that would be required to interpret the message. A RST message must echo the Message ID of the CON message, and must be empty.

2.3.4 Multicast

CoAP supports sending messages to multicast destination addresses. Such multicast messages must be NON messages.

2.4 Request/Response Semantics

CoAP operates under a similar request/response model as HTTP: a CoAP end-point in the role of a client sends one or more CoAP requests to a server, which services the requests by sending CoAP responses. Unlike HTTP, requests and responses are not sent over a previously established connection, but exchanged asynchronously over CoAP messages.

2.4.1 Requests

A CoAP request consists of the method to be applied to the resource, the identifier of the resource, a payload and Internet media type (if any), and optional meta-data about the request.

CoAP supports the basic methods of GET, POST, PUT, DELETE, which are easily mapped to HTTP. They have the same properties of *safe* (only retrieval) and *idempotent* (you can invoke it multiple times with the same effects) as HTTP. The GET method is safe, therefore it must not take any other action on a resource other than retrieval. The GET, PUT and DELETE methods must be performed in such a way that they are idempotent. POST is not idempotent, because its effect is determined by the origin server and dependent on the target resource; it usually results in a new resource being created or the target resource being updated.

A request is initiated by setting the Code field in the CoAP header of a CON or NON message to a Method Code and including request information.

2.4.2 Responses

After receiving and interpreting a request, a server responds with a CoAP response, whch is matched to the request by means of a client-generated token.

A response is identified by the Code field in the CoAP header being set to a Response Code. Similar to the HTTP Status Code, the CoAP Response Code indicates the result of the attemp to understand and satisfy the request.

0	1	2	3	4	5	6	7
Class			Detail				

Figure 2.4: Response

The upper three bits of the 8-bit Response Code number define the class of response. The lower five bits do not have any categorization role; they give additional detail to the overall class. There are three classes:

- 2- Success: The request was successfully received, understood, and accepted.
- 4- Client Error: The request contains bad syntax or cannot be fulfilled.
- 5- Server Error: The server failed to fulfill an apparently valid request.

The response codes are designed to be extensible: Response Codes in the Client Error and Server Error class that are unrecognized by an end-point must be treated as being equivalent to the generic Response Code of that class. However, there is no generic Response Code indicating success, so a Response code in the Success class that is unrecognized by an end-point can only be used to determine that the request was successful without any further details.

As a human readable notation for specifications and protocol diagnostics, the numeric value of a response code is indicated by giving the upper three bits in decimal, followed by a dot and then the lower five bits in a two-digits decimal. E.g., "Not found" is written as 4.04.

Responses can be sent in multiple ways:

- Piggy-backed: In the most basic case, the response is carried directly in the ACK message that acknowledges the request (piggybacking).
- Separate: It may not be possible to return a piggy-backed response in all cases. For example, a server might need longer to obtain the representation of the requested resource than it can wait sending back the ACK message, without risking the client to repeatedly retransmit the request message. Responses to requests carried in a NON message are always sent separately.

- Non-Confirmable: If the request is a NON message, then the response should be returned in a NON message as well. However, an end-point must be prepared to receive a NON response in reply to a confirmable request, or a confirmable response in reply to a NON message request.

2.4.3 Request/Response Matching

Regardless of how a response is sent, it is matched to the request by means of a token that is included by the client in the request as one of the options along with additional address information of the corresponding end-point. The token must be echoed by the server in any resulting response without modification.

The exact rules for matching a response to a request are as follows:

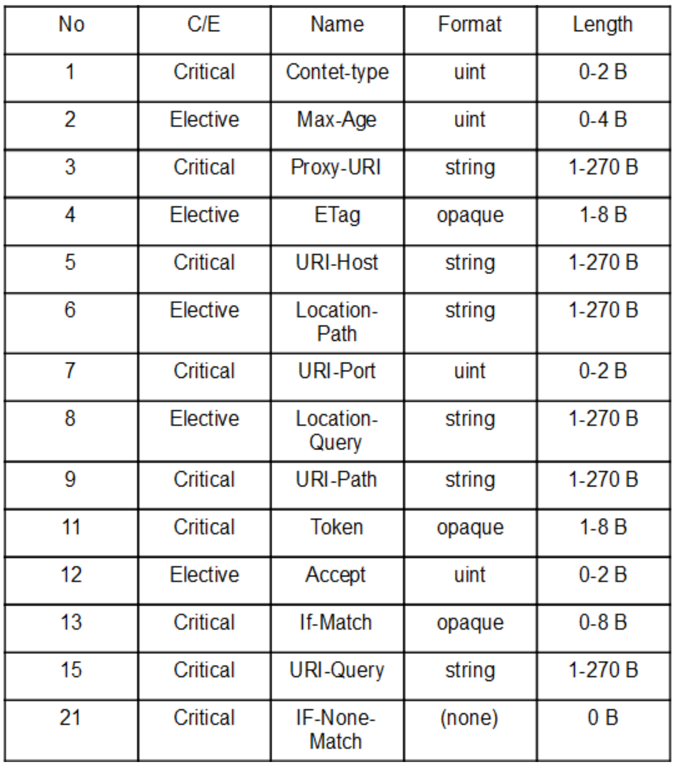
- For requests sent in a unicast message, the source of the response must match the destination of the original request. This is determined by the security mode used: with NoSec, the IP address and port number of the request destination and response source must match. With other security modes, in addition to the IP address and UDP port matching, the request and response must have the same security context.
- In a piggy-backed response, both the Message ID of the confirmable request and the acknowledgement, and the token of the response and original request must match. In a separate response, just the token of the response and original request must match.

The client should generate tokens in a way that tokens currently in use for a given source/destination pair are unique (a client can use the same token for any request if it uses a different source port number each time). An end-point receiving a token must treat it as opaque and make no assumptions about its format.

In case a CON message carrying a response is unexpected (a client might not be waiting for a response with the specified address and/or token), the confirmable response should be rejected with a RST message and must not be acknowledged.

2.4.4 Options

Both request and response may include a list of one or more options. For example, the URI in a request is transported in several options, and metadata that would be carried in an HTTP header in HTTP is supplied as options as well.



No	C/E	Name	Format	Length
1	Critical	Contet-type	uint	0-2 B
2	Elective	Max-Age	uint	0-4 B
3	Critical	Proxy-URI	string	1-270 B
4	Elective	ETag	opaque	1-8 B
5	Critical	URI-Host	string	1-270 B
6	Elective	Location-Path	string	1-270 B
7	Critical	URI-Port	uint	0-2 B
8	Elective	Location-Query	string	1-270 B
9	Critical	URI-Path	string	1-270 B
11	Critical	Token	opaque	1-8 B
12	Elective	Accept	uint	0-2 B
13	Critical	If-Match	opaque	0-8 B
15	Critical	URI-Query	string	1-270 B
21	Critical	IF-None-Match	(none)	0 B

Figure 2.5: CoAP Options.

CoAP defines a single set of options that are used in both requests and responses that can be seen in figure2.5.

Critical/Elective

Options fall into one of two classes: "critical" or "elective". The difference between these is how an option unrecognized by and end-point is handled (neither class is mandatory):

- Upon reception, unrecognized options of class "elective" must be silently ignored.
- Unrecognized options of class "critical" that occur in a CON request must cause the return of a 4.02 (Bad Option) response.
- Unrecognized options of class "critical" that occur in a CON response should cause the response to be rejected with a RST message.
- Unrecognized options of class "critical" that occur in a NON message must cause the message to be silently ignored.

2.4.5 Payload

Both requests and responses may include payload, depending on the method or response code respectively. Methods with payload are PUT and POST, and the response codes with payload are 2.05 (Content) and the error codes.

The payload of PUT, POST and 2.05 (Content) is typically a resource representation. Its format is specified by the Internet media type given by the Content-Type Option. No default value is assumed in the absence of this option.

2.01 (Created), 2.02 (Deleted), 2.04 (Changed) may include payload that is describing the result of the action. Again, the format of this payload is specified by the Internet media type given by the Content-Type Option; no default value is assumed in the absence of this option.

A response with a code indicating a Client or Server Error should include a brief human-readable diagnostic message as payload, explaining the error situation.

2.4.6 Caching

CoAP end-points may cache responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests.

The goal of caching in CoAP is to reuse a prior response message to satisfy a current request. In some cases, a stored response can be reused without the need for a network request, reducing the latency and network round-trips; a "freshness" mechanism is used for this purpose.

Freshness Model

When a response is "fresh" in the cache, it can be used to satisfy subsequent requests without contacting the origin server to provide an explicit expiration time in the future, using the Max-Age Option. The Max-Age Option indicates that the response is to be considered not fresh after its age is greater than the specified number of seconds.

2.4.7 Proxying

CoAP distinguishes between requests to an origin server and a request made through a proxy. A proxy is a CoAP end-point that can be tasked by CoAP clients to perform requests on their behalf. This may be useful, for example, when the request could otherwise not be made, or to service the response from a cache in order to reduce response time and network bandwidth or energy consumption.

CoAP requests to a proxy are made as normal CON or NON requests to the proxy end-point, but specify the request URI in a different way: The request URI in a proxy request is specified as a string in the Proxy-URI Option, while the request URI in a request to an origin server is split into the URI-Host, URI-Port, URI-Path and URI-Query Options.

When a proxy request is made to an end-point and the end-point is unwilling or unable to act as proxy for the request URI, it must return a 5.05 (Proxying Not Supported) response. If the authority (host and port) is recognized as identifying the proxy end-point, then the request must be treated as a local request.

Unless a proxy is configured to forward the proxy request of another proxy, it must translate the request as follows: The IP address and port of the origin server are determined by the authority component of the request URI, and the request URI is decoded and split into the URI-Host, URI-Port, URI-Path and URI-Query Options.

All options present in a proxy request must be processed at the proxy. Critical options in a request that are not recognized by the proxy must lead to a 4.02 (Bad Option) response being returned by the proxy. Elective options not recognized by the proxy must not be forwarded to the origin server. Similarly, critical options in a response that are not recognized by the proxy server must lead to a 5.02 (Bad Gateway) response. Again, elective options that are not recognized must not be forwarded.

If the proxy does not employ a cache, then it simply forwards the translated request to the determined destination. Otherwise, if it does employ a cache but does not have a stored response that matches the translated request and is considered fresh, then it needs to refresh its cache.

If the request to the destination times out, then a 5.04 (Gateway Timeout) response must be returned. If the request to the destination returns a response that cannot be processed by the proxy, then a 5.02 (Bad Gateway) response must be returned. Otherwise, the proxy return the response to the client.

If a response is generated out of a cache, it must be generated with a Max-Age Option that does not extend the Max-Age originally set by the server, considering the time the resource representation spent in the cache.

2.4.8 Method Definitions

In this section each method is defined along with its behavior. A request with an unrecognized or unsupported Method Code must generate a 4.05 (Method Not Allowed) response.

GET

The GET method retrieves a representation for the information that currently corresponds to the resource identified by the request URI. If the request includes one or more Accept Options, they indicate the preferred content-type of a response. If the request includes an ETag Option, the GET method requests that ETag be validated and that the representation be transferred only if validation failed. Upon success a 2.05 (Content) or 2.03 (Valid) response should be sent.

The GET method is safe and idempotent.

POST

The POST method requests that the representation enclosed in the request be processed. The actual function performed by the POST method is determined by the origin server and dependent on the target resource. It usually results in a new resource being created or the target resource being updated.

If a resource has been created on the server, a 2.01 (Created) response that includes the URI of the new resource in a sequence of one or more Location-Path Options and/or a Location-Query Option should be returned. If the POST succeeds but does not result in a new resource being created on the server, a 2.04 (Changed) response should be returned. If the POST succeeds and results in the target resource being deleted, a 2.02 (Deleted) response should be returned.

POST is neither safe nor idempotent.

PUT

The PUT method requests that the resource identified by the request URI be updated or created with the enclosed representation. The representation format is specified by the media type given in the Content-Type Option.

If a resource exists at the request URI the enclosed representation should be considered a modified version of that resource, and a 2.04 (Changed) response should be returned. If no resource exists then the server may create a new resource with that URI, resulting in a 2.01 (Created) response. If the resource could not be created or modified, then an appropriate error response code should be sent.

PUT is not safe, but idempotent.

DELETE

The DELETE method requests that the resource identified by the request URI be deleted. A 2.02 (Deleted) response should be sent on success or in case the resource did not exist before the request.

DELETE is not safe, but idempotent.

2.4.9 Response Code Definitions

Success 2.xx

This class of status code indicates that the clients request was successfully received, understood and accepted.

- 2.01 Created: Used in response to POST and PUT requests. If the response includes one or more Location-Path Options and/or a Location-Query Option, the values of these options specify the location at which the resource was created. Otherwise, the resource was created at the request URI. This response is not cacheable.
- 2.02 Deleted: Like HTTP 204 "No Content". This response is not cacheable. However, a cache should mark any stored response for the deleted resource as not fresh.
- 2.03 Valid: Related to HTTP 304 "Not Modified", but only used to indicate that the response identified by the entity-tag identified by the included ETag Option is valid. Accordingly, the response must include an ETag Option.
- 2.04 Changed: Like HTTP "No Content", but only used in response to POST and PUT requests. This response is not cacheable. However, a cache should mark any stored response for the changed resource as not fresh.
- 2.05 Content: Like HTTP 200 "OK", but only used in response to GET requests. This response is cacheable.

Client Error 4.xx

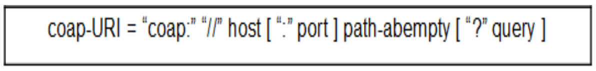
These response codes are applicable to any request method. The server should include a brief human-readable message as payload. Responses of his class are cacheable. They cannot be validated.

- 4.00 Bad Request: Like HTTP "Bad Request".
- 4.01 Unauthorized: The client is not authorized to perform the requested action.
- 4.02 Bad Option: The request could not be understood by the server due to one or more unrecognized or malformed critical options.
- 4.03 Forbidden: Like HTTP 403 "Forbidden".
- 4.04 Not Found: Like HTTP 404 "Not Found".
- 4.05 Method Not Allowed: Like HTTP 405 "Method Not Allowed", but with no parallel to the "Allow" header field.
- 4.06 Not Acceptable: Like HTTP 406 "Not Acceptable", but with no response entity.
- 4.12 Precondition Failed: Like HTTP 412 "Precondition Failed".
- 4.13 Request Entity Too Large: Like HTTP 413 "Request Entity Too Large".
- 4.15 Unsupported Media Type: Like HTTP 415 "Unsupported Media Type".

Server Error 5.xx

This class of response code indicates cases in which the server is aware that it has erred or is incapable of performing the request. These response codes are applicable to any request method. The server should include a human-readable message as payload. Responses of this class are cacheable. They cannot be validated.

- 5.00 Internal Server Error: Like HTTP 500 "Internal Server Error".
- 5.01 Not Implemented: Like HTTP 501 "Not Implemented".
- 5.02 Bad Gateway: Like HTTP 502 "Bad Gateway".



```
coap-URI = "coap:" "//" host [ ":" port ] path-abempty [ "?" query ]
```

Figure 2.6: *coap* URI scheme

- 5.03 Service Unavailable: Like HTTP 503 "Service Unavailable", but using the Max-Age Option instead of the "Retry-After" header field.
- 5.04 Gateway Timeout: Like HTTP 504 "Gateway Timeout".
- 5.05 Proxying Not Supported: The server is unable or unwilling to act as a proxy for the URI specified in the Proxy-URI Option.

2.5 CoAP URIs

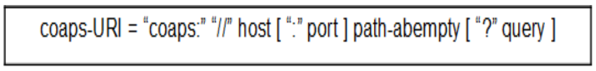
CoAP uses the *coap* and *coaps* URI schemes for identifying CoAP resources and providing means to locate the resource. Resources are organized hierarchically and governed by a potential CoAP origin server listening for CoAP requests (*coap*) or DTLS-secured CoAP requests (*coaps*) on a given UDP port. The CoAP server is identified via the authority of the generic syntax component, which includes a host identifier and optional UDP port number. The remainder of the URI is considered to be identifying a resource which can be operated on by the methods defined by the CoAP protocol. The *coap* and *coaps* URI schemes can thus be compared to the *http* and *https* URI schemes respectively.

2.5.1 coap URI Scheme

If the host is provided as an IP-literal or IPV4 address, then the CoAP server is located at that IP address. If the host is a registered name, then that name is considered an indirect identifier and the end-point might use a name resolution service, such as DNS, to find the address of that host. The host must not be empty. The port subcomponent indicates the UDP port at which the CoAP server is located. If it is empty or not given, then the defaul 5683 is assumed.

The path identifies a resource within the scope of the host and port. It consists of a sequence of path segments separated by a slash character (U+002F SOLIDUS "/").

The query serves to further parameterize the resource. It consists of a sequence of arguments separated by an ampersand character (U + 0026 AMPERSAND "&").



coaps-URI = "coaps:" "//" host [":" port] path-abempty ["?" query]

Figure 2.7: *coaps* URI scheme

The *coap* URI scheme supports the path prefix "/.wellknown/". This enables discovery of policy or other information about a host.

Application designers are encouraged to make use of a short, but descriptive URIs. As the environments where CoAP is used in are usually constrained for bandwidth and energy, the trade-off between these two qualities should lean towards the shortness, without ignoring descriptiveness.

2.5.2 coaps URI Scheme

All of the requirements listed for *coap URI Scheme* are also requirements for *coaps URI Scheme*. Unlike the *coap* scheme, responses to *coaps* identified requests are never public and thus must not be reused for shared caching. They can, however, be reused in a private cache if the message is cacheable by default in CoAP.

Resources made available via the *coaps* scheme have no shared identity with the *coap* scheme even if their resource identifiers indicate the same authority (the same host listening to the same UDP port). They are distinct name spaces and are considered to be distinct origin servers.

2.6 Resource Discovery

The discovery of resources offered by a CoAP end-point is extremely important in machine-to-machine applications where there are no humans in the loop and static interfaces result in fragility. A CoAP end-point should support the CoRE Link Format of discoverable resources. It is up to the server which resources are made discoverable (if any).

2.7 Default Ports

The CoAP default port number 5683 must be supported by a server for resource discovery and shoul be supported to provide access to other resources. The DTLS-secured CoAP default port number may be supported by a server for resource discovery and to provide access to other resources. In addition, other end-points may be hosted in the dynamic port space.

2.8 HTTP Mapping

CoAP supports a limited subset of HTTP functionality, and thus a mapping to HTTP is straighforward. There might be several reasons for mapping between CoAP and HTTP. for example when designing a web interface for use over either protocol or when realizing a CoAP-HTTP proxy.

There are two possible mapping via a forward proxy:

- CoAP-HTTP Mapping: Enables CoAP clients to access resources on HTTP servers through an intermediary. This is initated by including the Proxy-URI Option with an *http* or *https* URI in a CoAP request to a CoAP-HTTP proxy.
- HTTP-CoAP Mapping: Enables HTTP clients to access resources on CoAP servers through an intermediary. This is initiated by specifying a *coap* or *coaps* URI in the Request-Line of an HTTP request to an HTTP-CoAP proxy.

  Either way, only the Request/Response model of CoAP is mapped to HTTP. The underlying model of CON or NON messages, etc., is invisible and must have no effect on a proxy function. The following sections describe the handling of requests to a forward proxy. Reverse proxies are not specified as the proxy function is transparent to the client with the proxy acting as if it was the origin server.

2.8.1 CoAP-HTTP Mapping

This section specifies for any CoAP request the CoAP response that the proxy should return to the client. How the proxy actually satisfies the request is an implementation detail, although the typical case is expected to be the proxy translating and forwarding the request to an HTTP origin server.

Since HTTP and CoAP share the basic set of request methods, performing a CoAP request on an HTTP resource is not so different from performing it on a CoAP resource. The meaning of the individual CoAP methods when performed on HTTP resources are explained below.

If the proxy is unable or unwilling to service a request with an HTTP URI, a 5.05 (Proxying Not Supported) response should be returned to the client. If the proxy services the request by interacting with a third party (such as the HTTP origin server) and is unable to obtain a result within a reasonable time frame, a 5.04 (Gateway Timeout) response shold be returned; if a result can be obtained but is not understood, a 5.02 (Bad gateway) response should be returned.

- GET: The GET method requests the proxy to return a representation of the HTTP identified by the request URI. Upon success, a 2.05 (Content) response should be returned. The payload of the response must be a representation of the target HTTP resource, and the Content-Type Option be set accordingly. The response must indicate a Max-Age value that is no greater than the remaining time the representation can be considered fresh. If the HTTP entity has an entity tag, the proxy should include an ETag Option in the response and process ETag Options in requests as described below.

  A client can influence the processing of a GET request by including the following option:

  - Accept: The request may include one or more Accept Options, identifying the preferred response content-type.
  - ETag: The resquest may include one or more ETag Options, identifying responses that the client has stored. This requests the proxy to send a 2.03 (Valid) response whenever it would send a 2.05 (Content) response with an entity tag in the requested set otherwise.

- PUT: The PUT method requests the proxy to update or create the HTTP resource identified by the request URI with the enclosed representation.

  If a new resource is created at the request URI, a 2.01 (Created) response must be returned to the client. If an existing resource is modified, a 2.04 (Changed) response must be returned to indicate successful completion of the request.

- DELETE: The DELETE method requests the proxy to delete the HTTP resource identified by the request URI at the HTTP origin server.

  A 2.02 (Deleted) response must be returned to client upon success or if the resource does not exist at the time of the request.

- POST: The POST method requests the proxy to have the representation enclosed in the request be processed by the HTTP origin server. The actual function performed by the POST method is determined by the origin server and is dependent on the resource identified by the request URI.

If the action performed by the POST method does not result in a resource that can be identified by a URI, a 2.04 (Changed) response must be returned to the client. If a resource has been created on the origin server, a 2.01 (Created) response must be returned.

2.8.2 HTTP-CoAP Mapping

If a HTTP request contains a Request-URI with a *coap* or *coaps* URI, then the receiving HTTP end-point (called *the proxy* henceforth) is requested to perform the operation specified by the request method on the indicated CoAP resource and return the result to the client.

This section specifies for any HTTP request the HTTP response that the proxy should return to the client. How the proxy actually satisfies the request is an implementation detail, although the typical case is expected to be the proxy translating and forwarding the request to a CoAP origin server. The meaning of the individual HTTP methods when performed on CoAP resources are explained below.

If the proxy is unable or unwilling to service a request with a CoAP URI, a 501 (Not Implemented) response should be returned to the client. If the proxy services the request by interacting with a third party (such as the CoAP origin server) and is unable to obtain a result within a reasonable time frame, a 504 (Gateway Timeout) response should be returned; if a result can be obtained but is not understood, a 502 (Bad Gateway) response should be returned.

- GET: The GET method requests the proxy to return a representation for the CoAP resource identified by the Request-URI.

  Upon success, a 200 (OK) response should be returned. The payload of the response must be a representation of the target CoAP resource, and the Content-Type Option set accordingly. The response must indicate a Max-Age value that is no greater than the remaining time such that the representation can be considered fresh. If the CoAP entity has an entity tag, the proxy should include an ETag Option in the response.

  A client can influence the processing of a GET requet by including the following option:

  - Accept: Each individual Media-type of the HTTP Accept header in a request is mapped to a CoAP Accept option. HTTP Accept Media-type ranges, parameters and extensions are not supported by the CoAP Accept option. If the proxy cannot send a respone which is acceptable according to the combined Accept field value, then the proxy should send a 406 (not acceptable) response.
  - Conditional GETs: Conditional HTTP GET requests that include an *If-Match* or *If-None-Match* request-header field can be mapped to a corresponding CoAP request. The *If-Modified-Since* and *If-Unmodified-Since* request-header fields are not directly supported by CoaP, but should be implemented locally by a caching proxy.

- HEAD: The HEAD method is identical to GET except that the server must not return a message-body in the respone.

  Although there is no direct equivalent of the HEAD method of HTTP in CoAP, an HTTP-CoAP proxy responds to HEAD requests for CoAP resources, and the HTTP headers are returned without a mesage-body.

- POST: The POST method requests the proxy to have the representation enclosed in the request be processed by the CoAP origin server. The actual function performed by the POST method is determined by the POST method is determined by the origin server and dependent on the resource identified by the request URI.

  If the action performed by the POST method does not result in a resource that can be identified by a URI, a 200 (OK) or 204 (No Content) response must be returned to the client. If a resource has been created on the origin server, a 201 (Created) response must be returned.

- PUT: The PUT method requests the proxy to update or create the CoAP resource identified by the Request-URI the enclosed representation.

  If a new resource is created at the Request-URI, a 201 (Created) response must be returned to the client. If an existing resource is modified, either the 200 (OK) or 204 (No Content) response codes should be sent to indicate successful completion of the request.

- DELETE: The DELETE methods request the proxy to delete the CoAP resource identified by the Request-URI at the CoAP origin server.

A successful response should be 200 (OK) if the response includes an entity describing the status or 204 (No Content) if the action has been enacted but the response does not include an entity.

- CONNECT: This method cannot currently be satisfied by an HTTP-CoAP proxy function as TLS to DTLS tunneling has not been specified. It is however expected that such a tunneling mapping will be defined in the future. A 501 (Not Implemented) error should be returned to the client.

Chapter 3

Observe feature of CoAP (version 03)

Observe ([8]) is a feature of the CoAP protocol that aims to reduce the overall overhead in constrained networks caused by continous polling in some scenarios. The thesis focuses on Observe v.03 protocol, since it is the most time-accurate to CoAP v.08.

The protocol is based on the observer design pattern ([7]). In this design pattern, components, called *observers*, register at a specific, known provider, called *the subject*, that they are interested in being notified whenever the subject undergoes a change in state. The subject is responsible for administering its list of registered observers. If multiple subjects are of interest, an observer must register separately for all of them. The pattern is typically used when a clean separation between related components is required, such as data storage and user interface.

The observer design pattern is realized in CoAP as follows:

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

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

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

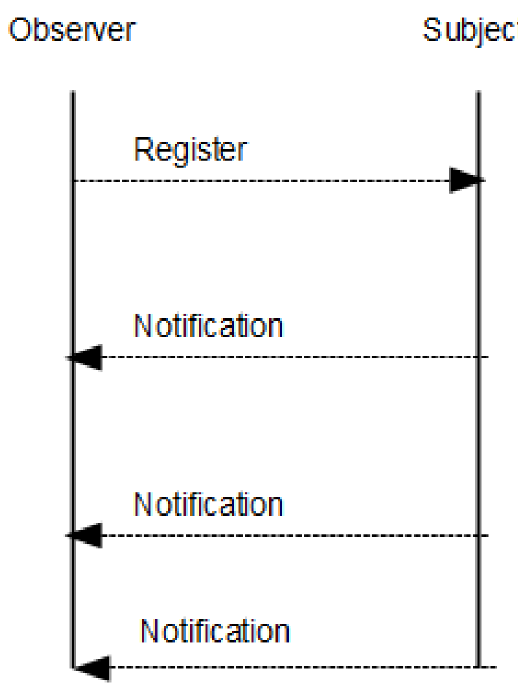
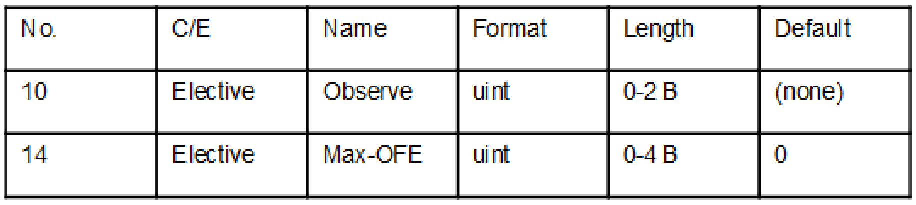


Figure 3.1: Observe design pattern

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

The client is removed from the list of observers when it is no longer interested in the observed resource. The server can determine the client's continued interest from the confirmable notifications acknowledgements. If a client wants to receive notifications after it has been removed from the list of observers, it needs to register again. The client can determine that it is still on the list of observers from the fact that it receives notifications. The protocol includes clear rules of what to do when a client does not receive a notification for some time, or a server does not receive acknowledgements.



No.	C/E	Name	Format	Length	Default
10	Elective	Observe	uint	0-2 B	(none)
14	Elective	Max-OFE	uint	0-4 B	0

Figure 3.2: Observe feature option

3.1 Options

3.1.1 Observe

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

In a response, the Observer Option identifies the message as a notification, which implies that the client has beed added to the list of observers and that the server will notify the client of further changes to the resoure state. The value of the option is a sequence number that can be used for reordering detection.

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

The Observe Option must not occur more than once in a request or response.

3.1.2 Max-OFE

The freshness of a notificaiton for caching purposes is determined by the Max-Age Option. However, a server may want to enable a cache to continue to optimistically use a cached representation even when the freshness indicated by the Max-Age Option has expired.

The time span for which this optimissm is justified is under control of the server: it can use the Max-OFE Option to indicate a desired *optimistic freshness extension*. This is also a promise by the server that it intends to send another notification within this time period. The exact semantics are defined in the sections below. The value of this option is a time span in seconds, measured from the Max-Age expiration time. The option is elective and defaults to zero (which means that no optimistic freshness extension is granted).

3.2 Client-side

3.2.1 Request

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

3.2.2 Notifications

Notifications are additional responses sent by the server in reply to the GET request. Each notification includes an Observe Option with a sequence number, a Token Option that matches the token specified by the client in the GET request, and a payload in the same representation format as the initial response.

A notification can be CON or NON. If a client does not recognize the token in a CON notification, it must not acknowledge the message and should reject it with a RST message. Otherwise, the client must acknowledge the message with an ACK message as usual.

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

Notifications will have a 2.05 (Content) response code in most cases. They may also have a 2.03 (Valid) response code, if the client includes an ETag Option in its request. In the event that the state of an observed resource is changed in a way that would cause a normal GET request to return an error, the server will send a notification with an error response code and empties the list of observers of the resource.

3.2.3 Caching

A client may store a notification like a response in its cache and use a stored response/notification that is fresh without contacting the origin server. A notification is considered fresh while its age is not greater than its Max-Age and if it has not been invalidated by a newer notification or as the request result.

Ideally, the server will provide a new notification exactly when the freshness of the latest notification expires. This may not always be possible though, due to network latency and/or resources that change their state in unpredictable intervals. In this case, the client may optimistically use a stale (non-fresh) notification while the notification's age is not greater than Max-Age plus Max-OFE and the notification has not been invalidated.

If the client does not receive a notification before Max-Age plus Max-OFE expires, the client can assume it has been removed from the list of observers. In this case, it needs to re-register by issuing a new GET request with an Observe Option.

To make sure it has a fresh representation and/or it is on the list of observers, a client may issue another GET request with an Observe Option at any time. The new GET request should specify a new token to avoid ambiguity. It is recommended that the client does not issue the request before the Max-Age of the latest notification expires.

When a client has one or more notifications stored, it can use the ETag Option in its request to give the server an opporunity to select a stored response to be used. The client may include an ETag Option for each stored response that is applicable. It needs to keep those responses in the cache until it is no longer interested in receiving notifications for the target resource or it issues a new GET request with a new set of entity-tags. When the observed resoure changes its state to a representation identified by one of the ETag Options, the server can send a 2.03 ("Valid") notification instead of a 2.05 ("Content") one.

3.2.4 Reordering

Messages that carry notifications can arrive in a different order than the original one. Since the goal is eventual consistency, a client can safely skip a notification that arrives later than a newer notification. For this purpose, the server sets the value of the Observe Option in each notification to a sequence number.

A client may treat a notification as outdated under the following conditions:

$$(V_1 - V_2)\%(2 * *16) < (2 * *15) \tag{3.1}$$

$$T_2 < (T_1 + (2 * *14)) \tag{3.2}$$

where V_1 is the value of the Observe Option of the latest valid notification received, V_2 the value of the Observe Option of the present notification, T_1 a client-local timestamp of the latest valid notification received (in seconds), and T_2 a client-local timestamp of the present notification.

The fisrt condition verfies that $V_2 > V_1$ holds in 16-bit sequence number arithmetic. The second condition checks that the time expired between the two incoming messages is not so large that the sequence number might have wrapped around and the first check is therefore invalid. (In other words, after about $2 * *14$ seconds elapse without any notification, the client does not need to check the sequence numbers in order to assume an incoming notification is new) . The constants of $2 * *14$ and $2 * *15$ are non-critical, as is the even speed or precision of the clock involved.

3.3 Server-side

3.3.1 Request

A GET request that includes an Observe Option requests the server not only to return a representation of the resource identified by the request URI, but also to add the client to the list of observers of the target resource. If no error occurs, the server must return a response with the representation of the current resource state and must notify the client of subsequent changes to the state as long as the client is on the list of observers.

A server that is unable to add the client to the list of observers of the target resource may silently ignore the Observe Option and process the GET request as usual. The resulting response must not include an Observe Option, the absence of which signals to the client that it will no be notified of changes to the resource state and needs to poll the resource instead.

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

Two requests relate to the same list entry if both the request URI and the source of the requests match. The source of a request is determined by the security mode used.

Any request with a method other than GET must not have a direct effect on a list of a resource observers. However, such a request can have the indirect consequence of causing the server to send an error notification which affects the list of observers.

3.3.2 Notification

A client is notified of a resource state change by an additional response sent by the server in reply to the GET request. Each such notification response must include an Observe Option and must echo the token specified by the client in the GET request. If there are multiple clients, the order in which they are notified is not defined; the server is free to use any method to determine the order.

A notification should have a 2.05 (Content) or 2.03 (Valid) response code. However, in the event that the state of a resource changes in a way that could cause a normal GET request to return an error (for example, if the resource is deleted), the server should notify the client by sending a notification with an appropriate error response code (4.xx/5.xx) and must empty the list of observers of the resource.

The representation format/media type used in a notification must be the same format used in the inital response to the GET request. If the server us unable to continue sending notifications in this format, it should send a 5.00 (Internal Server Error) notification and must empty the list of observers of the resource.

A notification can be sent as a CON or NON 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 messages; for resources that change infrequently, notifications can be sent in CON messages. The server can combine these two approaches depending on the frequency of state changes and the importance of individual notifications.

The acknowledgement of a CON notification implies the client's continued interest in being notified. If the client rejects a CON notification with a RST message, the server must remove the client from the list of observers.

3.3.3 Caching

The Max-Age Option of a notification should be set to a value that indicates when the server will send the next notification. For example, if the server sends a notification every 30 seconds, a Max-Age Option with value 30 should be included. The server may send a new notification before Max-Age ends. The server should also include a Max-OFE option so the client can continue to use a notification in case the next notification arrives a bit later due to network latency. If the client does not receive a new notification before Max-Age plus Max-OFE ends, it will assume that it was removed from the list of observers and may issue a new GET request to re-register its interest.

It may not always be possible to predict when the server will send the next notification, for example, when a resource does not change its state in regular intervals. In this case, the server should set Max-Age to a good approximation and Max-OFE to a time span for which the server is willing to keep the client in the list of observers.

Setting the values for Max-Age and Max-OFE is a trade-off between increased usage of bandwidth and the risk of stale information. Smaller values lead to more notifications and more GET requests, while greater values result in network or device failures being detect later and data becoming stale.

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 may send a 2.03 ("Valid") response with an appropriate ETag Option instead. The server must not assume that the recipient has any response stored other than those identified by the entity-tags in the most recent GET request.

3.3.4 Reordering

Because messages can get reordered, the client needs a way to determine if a notification arrived later than a newer notification. For this purpose, the server must set the value of the Observe Option in each notification to the 16 least-significatn bits of a strictly increasing sequence number. The sequence number may start at any value. the server must not reuse the same option value with the same client, token and resource wthin approximately $2 * *16$ seconds.

3.3.5 Retransmission

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

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

The server should use a number of retransmit attempts such that removing a client from the list of observers before Max-Age plus Max-OFE ends is avoided.

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

3.4 Intermediaries

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

To perform this task, the intermediary should make use of the protocol specified in this document, taking the role of the client and registering its own interest in the target resource with the next hop. If the next hop does not return a response with an Observe Option, the intermediary may resort to polling the next hop or may itself return a response without an Observe Option. Note that 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 type, and so on, must generate its own values for the Observe Option, and must set the values of the Max-Age Option and Max-OFE Option according to the age of the local current representation).

Because a client (or an intermediary in the client role) can only be once in the list of observers of a resource at a server (or an intermediary in the server role) as it makes no sense to observe the same resource multiple times, an intermediary must observe a resource only once, even if there are multiple clients for which it observes the resource.

3.5 Block-wise Transfers

Resources observed by clients may be larger than what they can be comfortably processed or transferred in one CoAP message. CoAP provides a block-wise transfer mechanism to address this problem. The following rules apply to the combination of block-wise transfer with notifications.

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 for all notifications/responses resulting from the GET request (until the client is removed from the list of observers or the server receives a new GET request from the client).

When sending a 2.05 (Content) notification, the server always sends all blocks of the representation, suitably sequenced by its congestion control mechanism, even if only some of the blocks have changed with respect to a previous value. The server performs the block-wise transfer by making use of the Block2 Option (see [5]) in each block. When reassembling representations that are transmitted in multiple blocks, the client must not combine blocks carrygin different Observe Option values, or blocks that have been received more than approximately 2**14 secons apart.

3.6 Discovery

A web link to a resource accessible by the CoAP protocol may indicate that the server encourages the observation of this resource by including the target attribute "obs".

Chapter 4

ThinkIP Software

The purpose of this chapter is to give an introduction of Patavina Technologies' ThinkIP. The software is a set of different protocols, generally used in new telecommunication technologies. The first section offers a brief description about the software architecture (see Fig. 4.1) and the second describes the module interfaces of each one of them. The main interest though, goes to the CoAP protocol and its implementation on the software package (explained in the third section).

4.1 Software Architecture

- Hardware Layer: This package exposes the *Hardware Abstraction Layer interface*, which provides a definition of an interface for each kind of peripheral in the system. It facilitates the porting of the software on other hardware platforms, by explicitly

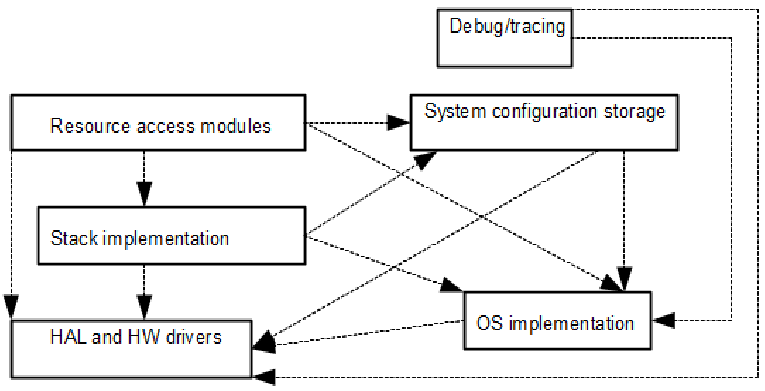


Figure 4.1: ThinkIP Architecture

defining the interactions the peripheral drivers are supposed to support.

- OS: This package exposes a basic operating system interface, capable of scheduling, executing and suspending on semaphores an arbitrary number of dynamically created tasks.

- System configuration storage: This package exposes the interface necessary to store, read, write and reset to factory defaults all the configuration parameters of the device.

- Debug utilities: This package provides a tracing utility which allows the developer to put log points throughout the code, and read out the log messages at run-time through the USB interface.

- Communication stack: This package provides the Resource Access Interface, which allows upper layers to register resource request handlers and issue resource requests, by implementing all the protocols required to bootstrap, use, and maintain secure wireless connectivity.

- Resource access: This package mainly uses the *Resource Access Interface* and the *Hardware Abstraction Layer* providing classes that respond to Resource requests by mapping them into the actions required on the device peripherals and state in general.

4.2 Module Interfaces

4.2.1 Hardware Abstraction Layer interface

This interface consists of a series of abstract classes, one for each peripheral type, defining a standard way of interacting with the peripheral.

4.2.2 OS interface

This interface allows a generic application relying on it to run on the different OS that implements such interface. Any OS implementing this interface is expected to comply to some some basic definitions and allow the minimal set of funtionalities defined in the following paragraphs.

Definition 1 *A task is an object that describes a sequence of CPU instructions to be executed when it is requested to "run".*

In general, for any OS implementation we assume the presence of a scheduler based on a *ready list* in which *ready* tasks are enqueued. Tasks are assigned priorities which specifies among many which task has the right to run first. When the right to run cannot be resolved by means of priority, it is automatically assigned to the oldest task.

The action of a task starting to execute, as a consequence of having been selected from the ready list by the OS, is defined as a *realization* of such task.

Given the above definitions, the OS Interface models the following functionalities and objects definitions:

- Post function: Allows a user application to inform the OS scheduler about tasks that need to change state and become *ready.* If the post function is invoked on a ready task it has no effect.
- Resume function: Allows suspended tasks to be re-inserted into the ready list (to restart running them from the last executed instruction).
- The information required by a task to be correctly processed by the OS is:
    1. An ID property, it uniquely identifies a realization of the task.
    2. A generic data pointer addressing optional data elements required by the task running routine.
    3. A task running routine, it is a function taking as single parameters the task generic data pointer. The execution of the task function of a given task with a given task parameter is a realization of that task. No concurrent realizations of the same task are allowed (e.g. an invocation of a task function with a given task argument is not allowed if the same task function with the same task argument is urrently being executed).
    4. A *auto_free* property, which is a boolean indicating wheter the task object should be freed after being executed.
    5. A priority propety. Among a set of ready tasks, the ones with highest priority run first.
    6. A suspend method. The suspend is a blocking call that freezes the task execution until the task is made ready again by someone else invoking the

resume function. After calling the suspend method a task is not ready anymore until the resume method is not called on that again. Suspension can only be invoked on a task that is running. As a consequence, for a single-core architecture the suspension is always invoked by the running task on itself. As soon as a running task selfsuspends, the scheduler chooses the next task to execute.

7. A state information describing the current state of a task.

- Definition of semaphores with the following primitives:

    1. Wait: suspends the task execution until a signal is invoked on the semaphore. If a signal primitive had previously been invoked the wait return without the task being suspended and the task can proceed straight with code execution.
    2. Signal: unlocks the first waiting task if a pending one exists or increments the counter of the signal calls for the future wait calls to run through without suspending the task that invoked it.

- Definition of mutexes: when there is no precedence to be managed among tasks and we just need to protect the access to a shared resource, we can avoid race conditions through the use of mutexes; the following primitives are defined for a mutex:

    1. Acquire: when invoked on a mutex from a task, if the mutex is not locked by another task, it is acquired (recursively) and a lock counter is increased.
    2. Release: when invoked on a mutex from a task that is locking it, a lock is removed. If the last lock is removed, the mutex can be locked again by the first task trying to acquire it.

- Definition of task message queues, which are objects characterized by:

    1. An enqueue method, taking a message parameter, that will be called by tasks willing to send a message to the queue.
    2. A dequeue method, returning a message value, which will be called by a task to get and remove a message from the queue.

- Definitions of message objects, which are objects characterized by a type property, which is an integer containing some task-specific value instructing the task on how to interpreter the message context.

4.2.3 System Configuration Interface

This interface consists in a class which defines the way of interaction with the *System Configuration Management*, which is in charge of storing and providing access to all the configuration parameters of the software.

The functionalities that are exposed through this interface are:

- Compile-time allocation of a configuration area, identified by an ID and initialized to a factory default value.
- Compile-time definition of a callback per configuration area, that should be invoked when the value stored in that area changes.
- Run-time methods providing access to the configuration areas (volatile memory) by ID.
- Run-time methods to
  - Restore factory defaults.
  - Load and save configuration from/to permanent memory.

4.2.4 Debug Interface

The functionalities that are exposed through this interface are:

- Compile-time definition of a log message type, which is a structure containing: ID and a Payload type
- Compile-time generation of a log method for each message type, taking one parameter which is a pointer to an instance of the associated payload type. This method will eventually store the message in a message queue that will be processed by a task that will output those messages through the USB interface.

4.2.5 Resource Access Interface

This interface defines the REST (REpresentational State Transfer) requests and response objects, and a process method.

Request

A REST request is defined by:

- A method (GET, POST, PUT, DELETE)
- A URI (uniform resource identifier)
- An optional payload

Response

A REST response is defined by

- A status code
- An optional payload with associated content type

The process method takes a request object as input and provides a response object as output.

A REST client will invoke the process method on the stack object providing the REST interface. This call is blocking and waits for a response from the stack, that is in charge of composing the appropriate CoAP request, sending it over the network, waiting the CoAP response and mapping into a REST response object that will be finally returned by the process method.

A REST server will instead implement the process method that will be invoked by the stack every time it receives a CoAP request, that will be mapped into a REST request object by the stack before being passed to the process method. The server is then in charge of composing a REST respose object and returning it to the caller, that will map it into a CoAP response that will be sent back to the client.

This way, the communication stack takes care of handling the transport of request and response objects through the network, while the exposed interface usage model is a simple method invocation.

4.3 ThinkIP CoAP

As said before, ThinkIP uses the version 08 of the CoAP protocol. To handle the protocol and divide the workload Patavina Technologies has adopted the following solution.

- Session Allocation
- Parse and Formatting
- Logic and Control

4.3.1 Session Allocation

It has the task to allocate a session between a client and a server without exceeding the established session limits. This event happens each time a new incoming request is handled or a new outgoing request is started.

4.3.2 Parse and Formatting

It has the task to retrieve all the necessary information from the CoAP packet that will be sent to the ″logic control". Viceversa, it also retreives all the necessary information from the "logic control" to create the CoAP packet.

4.3.3 Logic and Control

It manages the information retrieved in the *parse and formatting* phase. It identifies the source (client or server) and it takes the necessary steps to transmit the message to the other end-point. It also manages scenarios such as waiting for acknowledgement or losts packets.

ThinkIP uses a state machine diagram to represent the life of a CoAP session in *logic and control.*

Client side

The client side of the state machine diagram can be seen in figure 4.2, it handles the requests "sent" in CoAP. Once a session is established, the client enters the *IDLE* state, it checks the message type and treats it accordingly sending it to the upper layer.

If it is a NON message (*NON-Confirmable*), the client sets a timer, sends the request to the application and enters the *WAIT_SVR_RESP_NON* state where the possible scenarios are:

- The timer triggers, the client assumes that the request failed and ends the session.

- The client receives a NON message from the application, meaning that the request was successful. The client sets a new timer and enters the *ENDED* state, where it handles any lost packets until the timer triggers and the session ends.

If it is a CON (*Confirmable*) message, the client sets a timer and enters the *WAIT_ACK* state, the possibles scenarios are:

- The timer triggers, the client proceeds to retransmit the message and resets the timer staying on the same state. This event may repeat itself no more than a established maximum number of retransmissions, if such a number is reached the request failed and the session ends.

- Empty ACK received: the client receives an empty ACK, meaning that the server received the message but the application by some reason, was not able to deliver the expected data (the request is considered successful though). Therefore, the client sets a new timer and enters the *WAIT_SEPARATE* state, where it waits for the application to deliver the data. If the timer triggers the session ends, if instead it receives the expected data in a confirmable message the client sets a timer, transmits an ACK and enters the *CACHED* state.

- ACK Received: If the client receives a CON message (piggybacked ACK), the client sets a timer, transmits an ACK and enters the *CACHED* state. The other option is that the client receives a normal ACK (not piggybacked), so as before, the client sets a timer and enters the *CACHED* state.

The *CACHED* state handles any loose ends before ending the session. It manages any lost messages and retransmits the needed data until the timer triggers the session ending.

Server side

The server side of the state machine diagram can be seen in figure 4.3. It handles the requests "received" in CoAP. Analog to the client side, the server first enters the *IDLE* state, where it checks the message type, treats it accordingly and transmits it to the upper layer.
In the case of a NON message, the server sets a timer and enters the *WAIT_APP_RESP_NON* state, where it waits for the application to respond.

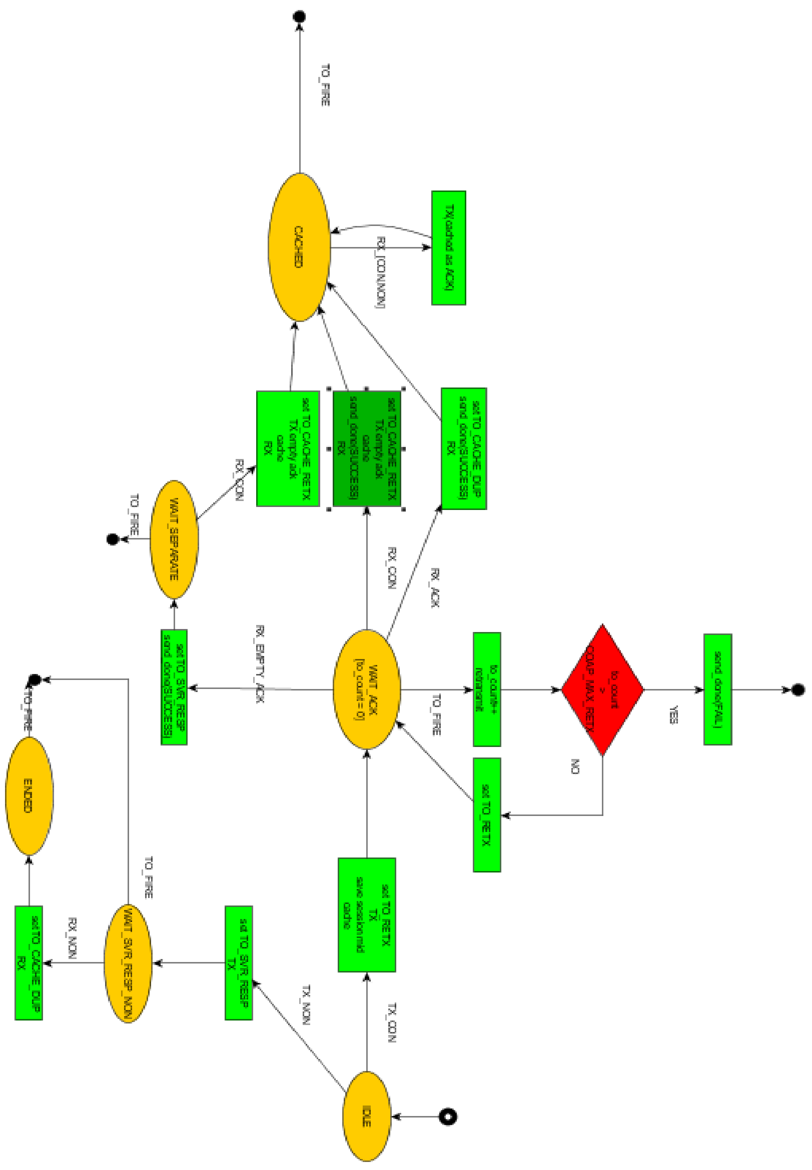


Figure 4.2: State Machine. Client side

Whether the application responds or not the server stays on the same state until the timer triggers the session ending.
In the case of a CON message, the server sets a timer and enters the *WAIT_APP_RESP_PB*, the possible scenarios are:

- The server receives a response from the application within the time limit, therefore it transmits a piggybacked ACK to the request, sets a new timer and enters the *CACHED* state.
- The timer triggers (not possible to transmit a piggybacked ACK), the server sets a new timer, transmits an empty ACK (*Empty ACK received* scenario in the client side) and enters the *WAIT_APP_RESP*. If the server receives a CON message it means that the empty ACK was lost, therefore it resets the timer and retransmits the empty ACK. If the timer triggers, the request failed and the session ends. If the application responds, the server sets a new timer and enters the *WAIT_ACK_SEPARATE*. The possible scenarios in this state are:
  - The timer triggers, the server retransmits the message, resets the timer staying on the same state. This event may repeat itself no more than an established maximum number of retransmissions, if such thing happens, the request fails and the session ends.
  - The server receives a CON message, meaning that the message with the data was lost, so the server proceeds to retransmit it and stays on the same state.
  - The server receives an empty ACK, meaning that the request was successful. The server sets a new timer and enters the *CACHED* state.

The *CACHED* state, as for the client side, handles any loose ends before ending the session. It manages any lost messages and retransmit the needed data until the timer triggers the session ending.

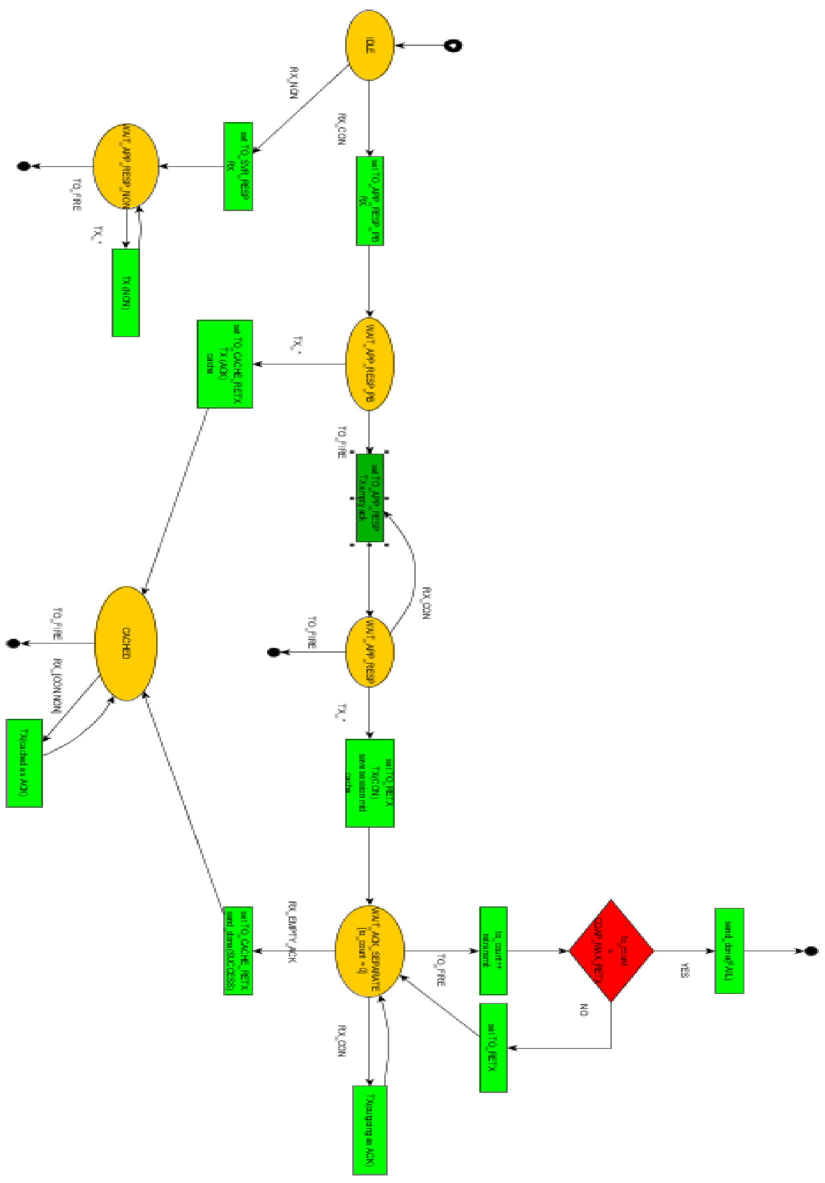


Figure 4.3: State Machine. Server side

Chapter 5

Observe implementation on ThinkIP

This chapter focuses on the observe feature implementation for the CoAP protocol in the ThinkIP (or ThinkIP-CoAP) software. Since ThinkIP uses the version 08 of the CoAP protocol we have implemented the version 03 of the observe feature.

The devices using this software are constrained devices, it was not possible to use some of the C++ standard libraries (such as "list" and "vector"). Also, ThinkIP uses a "timer" to manage events instead of threads since the latter would be a very CPU-consuming task.

As said before (in Chapter 3), the observe feature can be implemented in two ways, the first where notifications are CON responses and the second where the notifications are NON responses, our implementation uses CON responses.

Furthermore, in the observe feature, once a client makes an observe request to a server, a session is allocated between the end-points until the client loses interest in the resource or the server is unable to continue with the notifications. We have decided for a different choice that is more efficient (at least with ThinkIP structure). When the client makes an observe request a session is allocated between the end-points, but it ends once the client has received the server response, and each time the server sends a new notification to the client, a new session is allocated (and it ends once the client receives the notification).

Therefore, we must interpret the state machine diagram differently. When a subscriber makes an observe request, a session is allocated and ends when the subscriber has received the expected response. Assuming the server added the subscriber successfully to the list of observers, once it sends a notification, a new session is allocated and the notification is considered as a request (notifications are responses though) by the "logic control", so the subscriber follows the server side of the state machine diagram.

To implement the observe feature, we had to work with the application layer of ThinkIP-CoAP alone. the steps taken were the following:

5.1 Parse Formatting

The first step taken to implement the observe feature was to add the "observe option" in the CoAP packet, Observe-03 requires the CoAP packet to have the "Observe option" (It modifies the GET method, requesting the server to add the client to the list of observers) and the "Max-OFE option" (It indicates the freshness of a notification). So the necessary steps were:

- Add new variables: We added two unsigned integers variables in the interface of the CoAP packet to represent the observe options and a boolean variable (to acknowledge an observe option in a CoAP packet).
- Reception: Scenario where a received CoAP packet contains an observe option. The neccessary code for the *option fetching cycle* (part of the code in charge of fetching the diverse options in the CoAP packet) was added, such that ThinkIP-CoAP could recognize the new options and decode them before sending them to the "logic control" layer. Also, the boolean variable mentioned above is false by default, it changes its value once an observe option is identified within the CoAP packet.
- Transmission: Scenario when a CoAP packet with an observe option must be transmitted. The necessary code to retreive the information from"logic control" and encode the observe options into the CoAP packet was added.

5.2 Session Allocation

It is not possible to know *a priori* if a client is making an observe request, so each time a new session between a client and a server is established, a subscriber and an observe server variable must be instantianted to cover the scenario of a possible observe communication.

5.3 observe_common.h

It is a header that contains the class *ID* used by both the observe server and observe client. This class contains the required variable to uniquely identify a device. It is structured in the following way:

- uint16_t port: Instantiates a variable for the port number of the device (client or server). It is public.
- ipv6_addr_ref_t address: Instantiates an IPv6 address of the device. It is public.
- The class also contains the *set/get* methods for the previous variables.

5.4 Observe_server.cpp

Before making the step in the logic part of the protocol, certain structures that enables the Observe feature must be defined. The file *Observe_server.cpp* concerns the server side of the observe feature. It defines the following classes:

5.4.1 Client class

A class to represent a client object, it contains the data needed for a server to manage its subscribers list.

- ID id: Instantiates an ID variable.
- uri_t uri: Instatiates a uri variable. It contains a vector with the URI data plus a variable indicating the length of such vector.
- uint32_t token: Instatiates a uint32_t variable for the token. A necessity because the observe feature requires the notifications to have the same token option of the first observe request made by the client, such token is lost each time the session ends. Therefore, we store the token value of the observe request in this variable to use in future notifications.
- The class also contains the necessary "get-set" methods to manage the previous variables.

5.4.2 Observe_server class

A class to represent the server object, it contains the data needed for a server to offer the observe feature service:

- NOTIFICATION_PERIOD_MS: A constant value for the notifications frequency in miliseconds.
- MAX_CLIENTS: A constant value for the maximum number of subscribers the server can handle.
- MAX_OFE: A constant value for the value "Max-OFE" option.
- OBSERVE_INCR: A constant value for the "observe sequence" increment in each notification.
- uint16_t serv_obs_seq: Instantiates a variable for the observe sequence value of the notifications.
- timer_alarm_t serv_alarm: It instantiates a timer_alarm_t variable for the timer needed to trigger the notifications.
- vector_t<Client> Clients: It instantiates a vector_t of "Client" elements to store the data from each observing client.
- comm_coap_session_alloc_lc_block_t* coap_ext: It creates a pointer to a comm_coap_session_alloc_lc_block_t variable, needed to create a new session.
- static void t_server_alarm_cbk(timer_alarm_t* serv_alarm): It assigns a server_alarm callback to the Observe_server object.
- bool vectorfull(void): It checks if the vector *Clients* is full.
- void create_client(ID* id, uri_t* uri, uint32_t token): It creates a client with the given ID, URI and token value.
- bool client_onlist(Client* cl): It checks if the given client is on the vector.
- void send_notification(ipv6_address_t* subscriber_address,uint16_t subscriber_port, uri_t* uri, uint16_t payload_len, uint8_t* payload,uint16_t obs_seq, uint32_t token): It creates a notification for the given client with the requested data.

- void t_server_alarm_cbk(): It calls the method above for each of the observing clients everytime the timer triggers.
- Observe_server(comm_coap_session_alloc_lc_block_t* coap_ext): It is the constructor for the Observe_server class.
- void start_notifications(): It creates and starts the timer for the transmission of notifications.
- void stop_notifications(): It stops the timer for the transmission of notifications.
- void observe_server_block_receive(comm_pkt_coap_t* pkt_coap, comm_pkt_udp_t* pkt_udp, comm_pkt_ipv6_t* pkt_ipv6, comm_receive_cbk_t callback): It is used every time the server receives a CoAP packet, it checks if it is a packet regarding the observe feature and, if so, it treats it according to the observe rules and recommendations (**??**).
- Client* find(Client* cl): It checks if the vector contains the given client and if so, it return a pointer to such client.
- void removeclient(Client* cl): It removes the given client.
- void removeclient(uint16_t port, ipv6_address_t* address): It removes the given client.

5.5 Observe_client.cpp

The file *Observe_client.cpp* concerns the subscriber side of the observe feature, its structure is very similar to that of the previous file. It uses the following global variables and methods:

5.5.1 Observing_servers class

This class is used to represent the servers observed by the subscriber. It contains

- ID id: Instantiates an ID variable to identify the observed server.
- uint32_t token: Instantiates an uint32_t variable for the token. Since our implementation of the observe protocol creates a new session each time a notification is transmitted, the token value of the observe request must be stored in order to link a notification to a previous request.

- uint32_t MAX_OFE: Instantiates an uint32_t variable for the value of the "max-OFE" option.
- uint16_t current_obs_seq: Instantiates an uint16_t variable for the last "observe-sequence" value obtained from the last notification. It is used to order the notifications sequence (a fresh notification may arrive before an old one due to network latency).
- uint64_t current_time_stamp: Instantiates an uint64_t variable for the value of a local "time-stamp". It has a similar purpose to the "current_obs_seq" variable (notification ordering).
- timer_alarm_t client_alarm: It instantiates a timer_alarm_t variable for the timer needed to trigger the expiration of an observed resource.
- The class also contains the needed "get-set" methods to obtain and modify the previous variables.
- void t_client_alarm_cbk(timer_alarm_t* client_alarm): It assigns a client_alarm callback to the Observing_servers object.
- void start_timer(): It creates and starts a timer for a server to keep track of the last notification freshness.
- void reset_timer(): It resets the value of the started timer. It is used everytime a fresh notification is received

5.5.2 observing_client class

This class is used to represent the subscriber object. It contains the data needed for the subscriber to offer the observe feature:

- vector_t<Observing_servers> servers: It instantiates a vector containing "Observing_servers" elements. It stores the servers observed by the subscriber.
- uint16_t request_port: It instantiates a uint16_t variable for the destination port number in an outgoing observe request. Once the subscriber receives a response, it checks if the source port in the response matches the one in the request (to avoid acknowledging responses from a different server).

- ipv6_addr_ref_t request_address: It instantiates a ipv6_addr_ref_t variable for the destination IPv6 address in an outgoing observe request. It has the same purpose as the "request_port" variable, both variables are needded to uniquely identify a server.
- void t_client_alarm_cbk: It is invoked if the timer of one of the observed servers triggers. It removes the servers from the list of observed servers.
- bool server_onlist(Observing_servers* server): It checks if the given server is on the list of observed servers.
- Observing_servers* find(Observing_servers* server): It controls if the given server is on the list of the observed servers, if so, it returns a pointer to such server.
- Observing_servers* find(ID* id): It controls if there is a server with the given ID on the list of observed servers, if so, it returns a pointer to such server.
- void add_server(Observing_servers* serv): It adds the given server to the list of observed servers.
- void remove_server(Observing_servers* serv): It removes the given server from the list of observed servers.
- void observing_client_block_send(comm_pkt_coap_t* pkt_coap, comm_pkt_udp_t* pkt_udp, comm_pkt_ipv6_t* pkt_ipv6): It checks each outgoing request of a client. If it contains an observe option, it means is an observe request (and the client is also a subscriber), so the destination port and IPv6 address are stored in order to control the expected response. The URI value is also stored for the scenario in which the outgoing request does not contains an observe option, the subscriber checks if the destination server is on the list of observed servers, if so, it means the subscriber is not interested in the resource anymore and the entry for the server linked to the stored URI is removed from the subscriber lists.
- void observing_client_response_receive(comm_pkt_coap_t* pkt_coap, comm_pkt_udp_t* pkt_udp, comm_pkt_ipv6_t* pkt_ipv6): It checks if the received response contains an observe option and matches a previous observe request, if so, it means the server successfully added the subscriber to its list of

observers, therefore, it adds the server to the list of observed servers. It also stores the value of the token in the response needed to match future notifications.

- void manage_notifications(comm_pkt_coap_t* pkt_coap, comm_pkt_udp_t* pkt_udp, comm_pkt_ipv6_t* pkt_ipv6, uint64_t tstamp, comm_receive_cbk_t callback): It checks if the notification matches any of the observed servers, if so, it then controls if it is a fresh notification, if so, it updates the values of the "current_obs_seq", "current_time_stamp" and "MAX_OFE" variables and transmits the CoAP packet to the application.

5.6 Logic Control

Once the packet has been parsed, the data is passed to the "logic control" layer. The behaviour of a CoAP packet can be represented by ThinkIP CoAP state machine diagram. Such diagram can be divided in two parts, the client and the server side, before entering either of them, we are interested in knowing the source of a packet (client or server). Before the observe feature implementation, such knowledge was unnecessary since the state machine did not take into account such scenario, therefore we must first identify the nature of the source by checking the previous state of the packet source.

In the observe feature, once a client makes an observe request to a server, a session is allocated between the end-points until the client loses interest in the resource or the server is unable to continue with the notifications. We have decided for a different choice that has a better memory-efficiency (at least with ThinkIP structure). When the client makes an observe request, a session is allocated between the end-points, but it ends once the client has received the server response and each time the server sends a new notification to the client, a new session is allocated (and it ends once the client has received the notification message).

Therefore, we must interpret the state machine diagram differently. When a subscriber makes an observe request, a session is allocated and ends when the subscriber has received the response. Assuming the server added the subscriber successfully to the list of observers, once it sends a notification, a new session is allocated and the notification is considered as a request (notifications are responses) by the "logic control", so the subscriber follows the server side of the state machine diagram.

Furthermore, we had to modify the state machine diagram since it did not consider the scenario of a CON response (nature of the observe notifications), therefore the

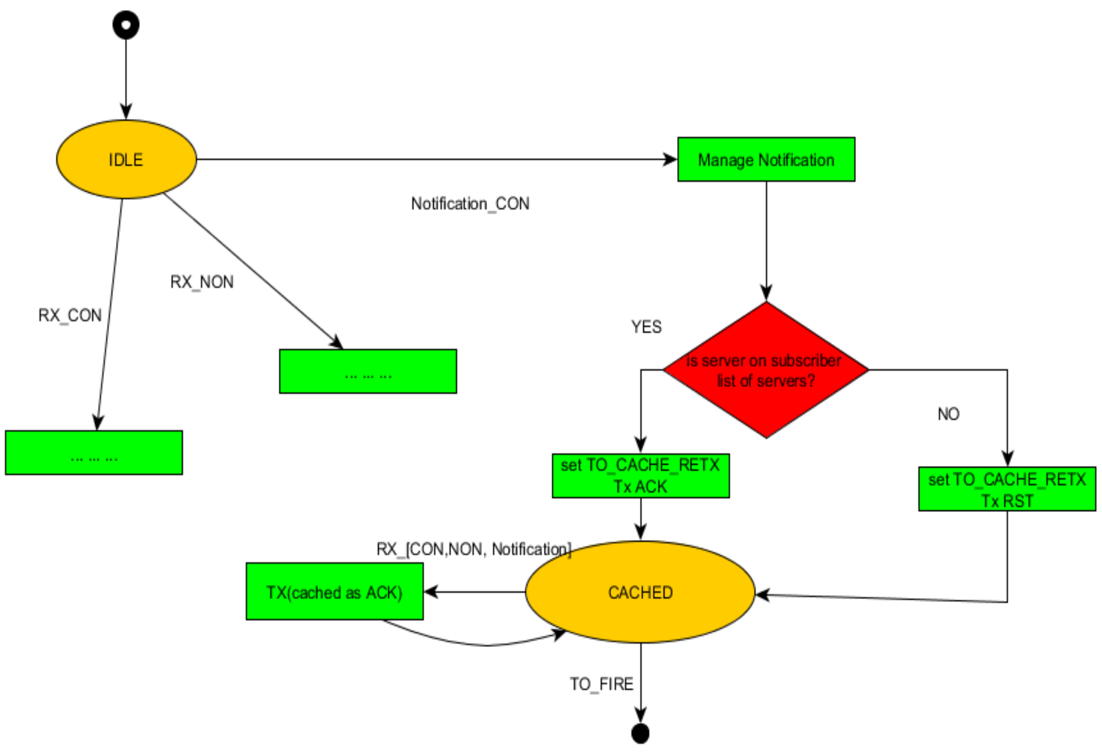


Figure 5.1: Modified State Machine Diagram

"server-side" diagram was modified (see Fig. 5.1) in order to manage these notifications.

5.6.1 Server Side

In this section, we will mention the "logic control" parts that have been altered to implement the observe feature for the server component. Even though the server acts as a client when transmitting the notifications, it was not necessary to modify the current code.

Block reception

It handles any request in reception, process the data and changes the state machine to the next logical state. With the observe feature, each request is analyzed to verify if it has "observe" data, if so, it is treated accordingly to the observe documentation. In our implementation, the subscriber also acts as a server when handling responses, so the needed code to identify the nature of the source was included.

Timer triggers

There are different points in the state machine (WAIT_APP_RESP, WAIT_ACK_SEPARATE state and WAIT_APP_RESP_PB) where it is possible for the timer to trigger and end the session. Since the absence of an ACK indicates the disinterest of a client for a resource, the code was modified such that when these events trigger, the server removes the subscriber from the observe entry list.

Responses

Notifications are managed by the "Observe_server" class and treated accordingly by the state machine. However, further consideration must be taken with the response to the "observe request", such response is a normal response to a GET request with the observe option included, so the code that builds the CoAP response to a GET request was modified in order to add the observe request response.

5.6.2 Subscriber Side

In this section, we will mention the "logic control" parts that have been altered in order to implement the subscriber side of the observe feature. The subscriber implementation regarding the server component is very similar to the previous server component, so it did not bring much complexity to the observe implementation.

Requests

Once the subscriber makes an observe request, it checks if the response contains an observe option, if so, the observe request was successful and the subscriber adds the server to the list of observed servers. This is the only component altered in the client side of the protocol to implement the observe feature.

ACKs

Similar to the server, once a subscriber receives a CoAP message, it checks if it is an observe response linked to one of the observed servers, if so, the data is handled accordingly to the observe documentation and the subscriber transmit an ACK for the notification. The current state machine diagram does not handles this scenario very well, so it was modified in order to treat with this scenario.

Once a subscriber receives a CON notification, it checks if it is linked to one of its observed servers, if so, it sends the data to the application, returns an ACK message and enters the *CACHED* state. If it is not linked to one of itss observed servers, it discards the packet, returns an RST message and enters the *CACHED* state.

5.7 Numerical Results

In this section, we present the new memory usage with the addition of the new code.

- Before:
  - ROM: 214342 (207342 + 7000) Bytes.
  - RAM: 10618 (7000 + 3618) Bytes.

  Where 207342 bytes is due to the ROM usage, 7000 bytes for RAM usage and 3618 bytes by the bss section.

- After:
  - ROM: 219928 (212802 + 7126) Bytes.
  - RAM: 10801 (7126 + 3675) Bytes.

  Therefore, we have 5460 more bytes from ROM usage, 126 bytes for RAM usage and 57 bytes for the bss section due to the observe feature implementation.

The whole code was written in the C++ language using "Eclipse" ([2]) java IDE. Moreover, in order to test it, we have used the following software:

- Copper ([1]): It is a Firefox plug-in that serves as a CoAP client simulator.
- Wireshark ([4]): Software that analyze packets, needed for the packets transmitted in the simulation.

Chapter 6

Conclusion

In this thesis we have illustrated the CoAP (version 08) protocol and the observe feature (version 03) implementation of it in the first two chapters. It is a fairly new protocol that could have great successl if the field of constrained networks continues to extend its market. The observe feature is an extension to CoAP that solves some shortcomings in certain scenarios (such as polling) eliminating unnecessary GET requests to nodes, reducing the overall network overhead and therefore improving the network efficiency.

We have also illustrated the implementation of the CoAP protocol (version 08) in the ThinkIP software of Patavina Technologies. Currently, such implementation is working as expected by the documentation (in Chapter 3).

To be more memory-efficient (at least in ThinkIP software), each notification creates a new session between the subscriber and the server.

Currently, ThinkIP implements CoAP version 08. The protocol is still under development and there is not yet a final version, once the CoAP protocol presents a final version or its current version has major changes, ThinkIP plans to update its CoAP version and consequently update its observe feature too.

Regarding the observe feature in the current ThinkIP code, it is more memory-efficient to create a new session for each notification. In our implementation, responses are treated similar to request, therefore, the client acts like a server when dealing with notifications, furthermore, we had modified the state machine diagram in the "logic control" to deal with this scenario (responses must be treated differently to requests).

In future works, ThinkIP will try to find an alternative solution to the current session memory allocation method, such that only one session will be needed when initiating an observe communication. This change would not only be more faithful to the current observe documentation, but it would also help the code indipendency of observe within CoAP (e.g. we can avoid the new state included in the "logic control" component of CoAP).

Bibliography

[1] Copper. `http://people.inf.ethz.ch/mkovatsc/copper.php`.

[2] Eclipse. `http://www.eclipse.org/`.

[3] Patavina technologies. `http://www.patavinatech.com/`.

[4] Wireshark. `http://www.wireshark.org/`.

[5] C. Bormann and Z. Shelby. Blockwise transfers in coap, draft-ietf-core-block-04 (work in progress). Technical report, July 2011.

[6] Sensinode K. Hartke Universitaet Bremen TZI B. Frank SkyFoundy C. Bormann, Z. Shelby. Constrained application protocol (coap), draft-ietf-core-coap-08 (work in progress). Technical report, November 2011.

[7] R. Johnson E. Gamma, R. Helm and J. Vlissides. Design patterns: Elements of reusable object-oriented software, November 1994.

[8] Z. Shelby Sensinode K. Hartke, Universitaet Bremen TZI. Observing resources in coap, draft-ietf-core-observe-03 (work in progress). Technical report, October 2011.

[9] L. Richardson and S. Ruby. Restful web services, May 2007.

List of Figures