Escola Tècnica Superior d'Enginyeria de Telecomunicació de Barcelona

UNIVERSITAT POLITÈCNICA DE CATALUNYA

Projecte final de carrera

RESTful Web services in Wireless Sensor Networks

Estudis: Enginyeria de Telecomunicació

Autor: Pol Moreno Yeste

Directora: Anna Calveras

Any: 2011

Contents

Resum

Les xarxes de sensors inalàmbriques s'han tornat molt populars durant els darrers anys. Estan formades per sensors autònoms que estan distribuïts per monitoritzar les condicions físiques i ambientals i transferir de forma col·laborativa les seves dades a través de la xarxa fins a un punt central. Aquestes xarxes faciliten la creació i integració de xarxes de baix cost en escenaris on utilitzar cablejat és difícil i costós. Alguns exemples d'àmbits on aquestes xarxes es podrien aplicar serien la domòtica, la logística o la salut.

El principal objectiu d'aquest treball és investigar i proposar una solució per aconseguir integrar serveis Web sobre xarxes de sensors inalàmbriques. Per a realizar aquesta tasca, s'ha fet una recerca per saber quin dels models d'arquitectures actuals més utilitzats a la Web (REST o SOAP) és el que s'ajusta millor als requeriments d'aquestes xarxes. Les conclusions obtingudes apunten a que el model REST és el que millor s'ajusta.

El següent pas ha estat triar un protocol d'aplicació per les diferents xarxes de sensors. El protocol utilitzat en aquest treball és el protocol CoAP. S'ha desenvolupat una programari per integrar aquest protocol al sistema operatiu TinyOS, que s'ha utilitzat i provat sobre el maquinari TelosB.

El protocol d'aplicació més utilitzat a Internet és el protocol HTTP. Per aconseguir integrar la solució proposada en el model actual d'Internet, és necessària una traducció entre el protocol CoAP i el protocol HTTP. En aquest treball es proposa una solució (tant maquinari com programari) basada en un proxy que permet als clients HTTP actuals accedir a recursos CoAP.

Finalment, s'ha realitzat una avaluació per extreure algunes conclusions sobre les implementacions realitzades. Els resultats mostren que l'ús d'aquest protocol dóna millors resultats en termes de latència si es compara amb utilitzar HTTP/1.0 sobre dispositius limitats en recursos. També s'ha realizat una comparativa entre utilitzar un client CoAP per accedir als recursos de la xarxa o bé fer servir un client HTTP per connectar-se mitjaçant un proxy. Els resultats mostren que en els casos on fer servir un client CoAP no sigui possible, la solució proposada és una bona alternativa que afegeix poca latència.

Resumen

Las redes de sensores inalámbricas se han vuelto muy populares durante los últimos años. Están formadas por sensores autónomos que están distribuidos para monitorizar las condiciones físicas y ambientales y transferir de forma colaborativa sus datos a través de una red hasta un punto central. Estas redes facilitan la creación e implantación de redes de bajo coste en lugares donde emplear cableado es difícil y costoso. Algunos ámbitos en los que se podrían aplicar estas redes serían la domótica, la logística o la salud.

El principal objetivo de este trabajo es investigar y proponer una solución para conseguir integrar servicios Web en redes de sensores inalámbricas. Para realizar esta tarea, se ha investigado cuál de los dos modelos de arquitecturas Web actuales más utilizadas (REST o SOAP) es el que mejor se ajusta a las características de estas redes. Las conclusiones obtenidas apuntan a que el modelo REST es el que mejor se ajusta.

El siguiente paso ha sido seleccionar un protocolo de aplicación para las diferentes redes de sensores. El protocolo utilizado en este trabajo es el protocolo CoAP. Se ha desarrollado un software para integrar este protocolo en el sistema operativo TinyOS, que se ha utilizado y probado sobre el hardware TelosB.

El protocolo de aplicación más utilizado en Internet es el propocolo HTTP. Para conseguir integrar la solución propuesta para las redes inalámbricas de sensores en el modelo actual de Internet, es necesaria una traducción entre los protocolos CoAP y HTTP. Este trabajo propone una solución hardware y software basada en la utilización de un proxy que permite a los clientes HTTP acceder a recursos CoAP.

Finalmente, se ha realizado una evaluación para extraer algunas conclusiones sobre las implementaciones realizadas. Los resultados muestran que el uso de este protocolo proporciona mejores resultados en términos de latencia si se compara con utilizar HTTP/1.0 sobre dispositivos con recursos limitados. También se ha realizado una comparativa entre utilizar un cliente CoAP para acceder a los recursos de la red o utilizar un cliente HTTP mediante el proxy propuesto. Los resultados muestran que en los casos en los que utilizar un cliente CoAP no sea viable, la propuesta de proxy es una buena solución que ofrece una baja latencia.

Abstract

Wireless Sensor Networks (WSNs) have become very popular in recent years. A WSN consists of distributed autonomous sensors to monitor physical or environmental conditions and to cooperatively send their data through the network to a main location. A WSN facilitates the creation of low-cost networks that can be used for multiple applications when the use of wires is impractical and expensive. Examples of these applications are home and building automation, asset management and logistics or health care automation.

The main objective of this work is to investigate and propose a solution to enable Web services in WSNs. To accomplish this task, a research to choose the Web architectural style that fits best with WSNs has been done. Discussions show that the REST architectural style seems to be the one which is more suitable for WSNs.

The next step is to choose an appropriate application layer protocol. In this sense, the CoAP protocol has been used to enable Web services in constrained networks. In this work, a CoAP implementation for the TinyOS operating system is proposed and developed.

The most popular application protocol used in the Internet is HTTP. In order to integrate the proposed solution with the Internet, some translation between CoAP and HTTP is needed. In this work, a hardware and software proxy-based solution to connect CoAP resources with the actual Internet is proposed and developed. This solution also enables extra services that are interesting in WSNs.

Finally, a performance evaluation is done in order evaluate the proposed implementation. Results show that using CoAP in constrained devices in WSNs improves the overall latency in comparison with using HTTP/1.0 in constrained nodes. A comparison between using a CoAP client to access CoAP resources and using a HTTP client using the proposed proxy solution has been done. Results show that using the proxy solution adds low extra latency, thus providing a good solution when using a CoAP client is not possible.

Overview

Introduction

In some applications, using wired networks for monitoring the environment is usually impractical and expensive. For this reason, it is interesting to create low-cost network architectures that give mobility to its terminals. In this sense, the deployment of Wireless Sensor Networks (WSNs) is a good solution. A WSN consists of distributed autonomous sensors able to monitor physical or environmental conditions and to cooperatively send their data through the network to a main location. These networks are composed of hundreds of low-power and low-cost devices that are characterized by having constrained resources, limited operational capabilities and a short communication range. These constraints are the critical aspects that influence the choice of a protocol stack.

A widely-used protocol for the physical and link layers is IEEE 802.15.4[1]. When a WSN uses this standard, it is called a Low Power Wireless Personal Area Network (LoWPAN). IEEE 802.15.4 is nowadays a standard for low layers of WSNs. However, when interconnecting different WSNs or sensor nodes from different manufacturers many problems arise. The growing interest of these networks has led to the development of many solutions that limited the possibility to interconnect and integrate various WSNs. To overcome this problem, one solution is using an existing and well-known protocol such as IP. However, it was considered impractical because of the highly constrained requirements.

Recent developments prove that it is possible to enable efficient IPv6 communications over IEEE 802.15.4 links introducing an adaptation layer[2]. The resulting protocol stack goes under the name of IPv6 over Low Power Wireless Personal Area Networks (6LoWPAN).

Enabling IP on constrained devices has several benefits. WSNs can be connected to external IP networks without requiring intermediate gateways. Furthermore, all knowledge from IP-based networks can be reused, avoiding the creation of new tools for managing, configuring or diagnosing these networks. IP connectivity would also allow a crucial innovation in the Internet field. In this perspective, Internet would be a network with embedded objects that would be able to exchange information and interact with their environment. This new concept is referred as the so-called *Internet of Things*.

A key aspect to completely integrate these networks is to extend the actual Web architecture to WSNs. Furthermore, sensor nodes can be treated as any other Web resource that would be accessed using standard Web mechanisms. This new approach is known as *Web of Things*.

The implementation of Web services in these networks must be based on an architectural style adapted to WSNs requirements. Also, the implementation should reuse and adapt existing protocols and avoid the creation of new ones in order to avoid interoperability problems.

Motivation

The main motivation of this work is to investigate and propose a solution to integrate Web services in WSNs.

Recently, some operating systems for constrained devices such as TinyOS[3] or Contiki[4] have been developed. Both provide a 6LoWPAN stack and libraries to handle protocols from upper layers such as UDP or TCP. These operating systems are open source, so they can be copied, used or modified freely. They are prepared to work in some hardware platforms such as TelosB[5] that allows a developer to easily test any program made for those operating systems. The existence of these tools simplifies the development and testing of software solutions for implementing Web services in constrained networks.

The actual Web communication is mainly based on the HTTP protocol. However, this protocol might not be the best solution for constrained networks because of the constrained requirements.

In conclusion, the main motivation of this work is to provide a generic software to easily integrate Web services for constrained devices, to use these Web services in more capable devices such as Personal Computers (PCs) or embedded computers and to provide a software solution for Web interoperability. Finally, a performance evaluation will be made to analyze if the resulting software is suitable for WSN requirements.

Objectives

The objectives of this work are the following:

- Understand the most important concepts of the 6LoWPAN stack.
- Analyze different architectural styles for integrating Web services in WSNs and decide which is the one that fits best.
- Search for a possible solution for the application layer to use on top of the 6LoWPAN stack. In this case, the Constrained Application Protocol (CoAP)[6] protocol is analyzed and discussed.
- Search for an existing software that implements the CoAP protocol for constrained devices or develop a new one. In this work, the TinyOS operating system is used. Thus, this software must be adapted for this operating system.
- Search for an existing software that implements the CoAP protocol for the Linux operating system or develop a new one.
- Use the previous software to propose and develop a solution for Web interoperability.
- Perform an evaluation of the resulting performance of the different software components created.

Document structure

This document is divided into the following sections:

Section 1 gives an overview of the 6LoWPAN stack introducing the most important concepts.

Section 2 gives the state-of-art of the current Web services. It discusses which architectural style (REST or SOAP) is more suitable for constrained environments.

Section 3 presents the CoAP application protocol for constrained devices. It explains how this protocol works and all its features.

Section 4 introduces the TinyOS operating system and explains some characteristics about its internal structure and programming style. Special attention is paid to its memory and concurrency model.

Section 5 explains different possibilities to integrate real world RESTful Web services with 6LoWPAN networks. It provides a solution for WSNs interoperability.

Section 6 presents a CoAP implementation for the TinyOS operating system. Hardware and software are described in this section.

Section 7 presents a new framework for C applications that allows developers to easily integrate CoAP server and client services into their programs.

Section 8 presents and describes a hardware and software solution for Web services integration.

Section 9 presents the performance evaluation results.

Section 10 presents conclusions and future directions.

1 6LoWPAN

1.1 Overview

The current Internet model is formed by the *core* and the *fringe Internet*. *Core Internet* is formed by backbone routers and servers including any type of network device. It changes rarely and has extremely high capacity. The majority of Internet nodes are part of the so-called *fringe Internet*. It includes all smart phones, laptops, personal computers and local network infrastructures[7].

The biggest Internet opportunity nowadays is in the Internet of Things. Internet of Things can be defined as the Internet that encompasses all the embedded devices and networks that are natively IP-enabled and the Internet services for monitoring and controlling those devices. Predictions say that the potential long-term size of the Internet of Things is of one trillion devices and the greatest potential growth comes from embedded, low-powered, wireless devices that form the Wireless Embedded Internet.

6LoWPAN was born to enable the Wireless Embedded Internet by simplifying IPv6 functionalities, defining very compact header formats and taking into account wireless communication characteristics. 6LoWPAN introduces IP-based technologies in low-powered wireless nodes to solve interoperability problems. The use of well-know IP-based technologies has some benefits:

- Sensors can be connected to external IP networks, reusing existing technologies.
- IP technology is specified in an open and free way, written in standard documents that are available to anyone.
- Tools for monitoring and managing IP-based networks already exist and are widely used.

Until recent years only powerful devices have been able to participate natively with the Internet. Enabling IP in low-powered wireless devices is particularly challenging because:

- **Devices are battery powered**. Battery-powered embedded devices must maintain duty cycles and power consumption low. This means that sometimes the devices can enter in *sleep* mode while IP assumes that the devices are always connected.
- **Multicast support**. Wireless technologies such as IEEE 802.15.4 do not usually support multicast.
- **Mesh topologies**. Wireless embedded radio technologies typically benefit from multi-hop and mesh features to increase the coverage area. This cannot be easily performed by IP routing protocols.
- **Bandwidth**. IEEE 802.15.4 has limited bandwidth and MTU of 127 bytes. The minimum frame size for IPv6 is 1280 bytes.

- **Reliability**. Failures can occur in wireless nodes because of they are asleep, out of battery or node failure.

In this perspective, the IETF CoRE working group[8] has been created and its aim is to extend the Web architecture to constrained networks and devices. The result is an efficient IPv6 extension applied in the wireless embedded domain that enables *end-to-end* IP networking and features for a wide range of embedded applications. Some applications can be:

- Home and building automation.
- Asset management and logistics.
- Smart metering.
- Healthcare automation.
- RFID infrastructures.

1.2 6LoWPAN network architecture

The Wireless Embedded Internet is accomplished by connecting *stub networks*, which are networks that do not act as transit to other networks. A 6LoWPAN is a collection of embedded wireless nodes with shared common IPv6 prefix. There are three types of 6LoWPANs that can be seen in figure 1.1:

- **Ad hoc 6LoWPAN**. It is not connected to the Internet, but instead operates without an infrastructure.
- **Simple 6LoWPAN**. It is connected to other external IP networks through a 6LoWPAN edge router. For this purpose, both a *backhaul link* and a *backbone link* can be used.
- **Extended 6LoWPAN**. It contains several 6LoWPANs of multiple edge routers connected through a backbone link.

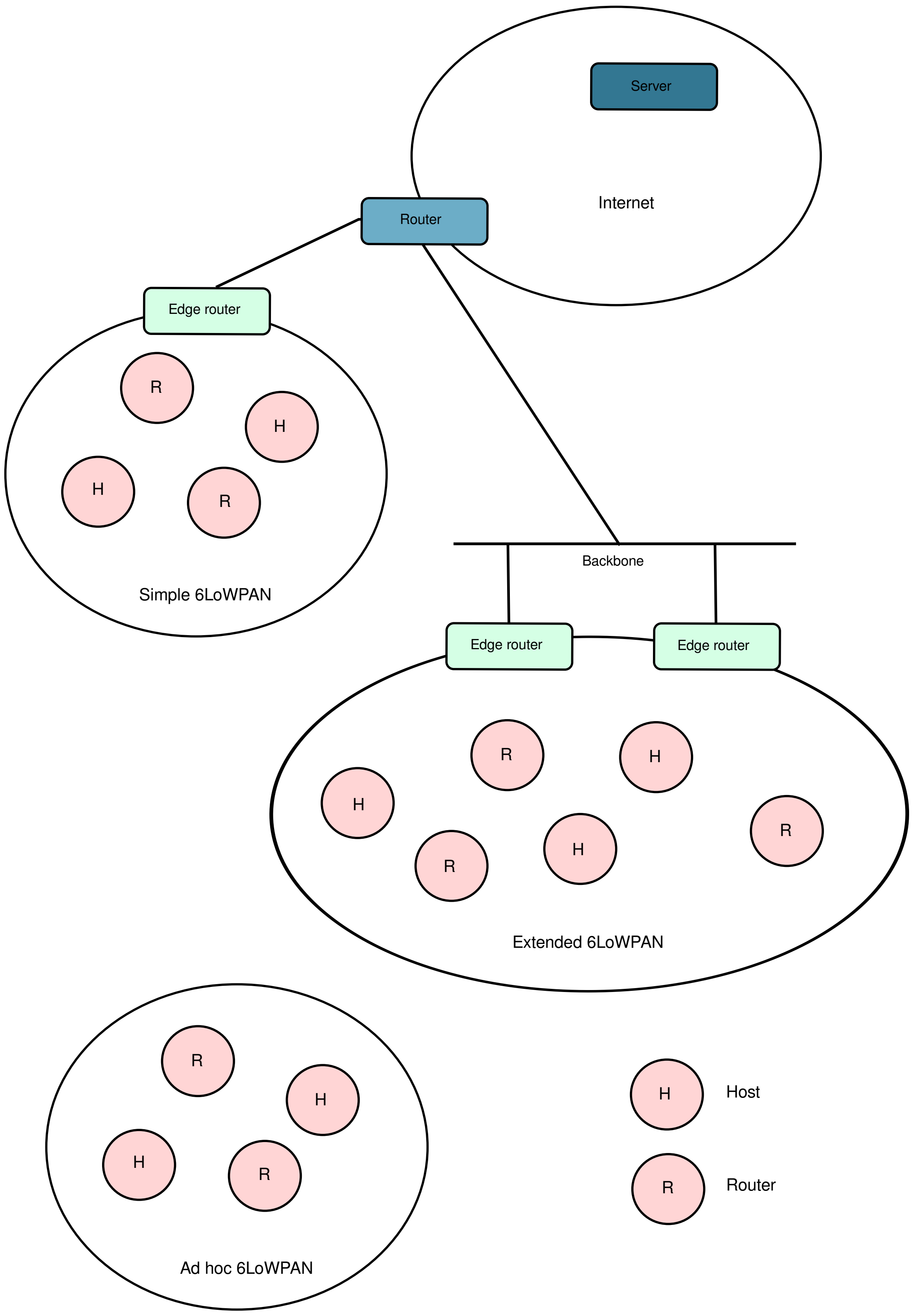


Figure 1.1: 6LoWPAN architecture

6LoWPAN networks are connected to other IP networks through edge routers. They route the traffic in and out the 6LoWPAN and they also handle 6LoWPAN compression and Neighbor Discovery. Neighbor Discovery (ND) is very important because defines how hosts and routers interact with each other on the same link.

6LoWPAN nodes are free to move through the LoWPAN, between edge routers and between 6LoWPANs. A multi-hop mesh topology within the 6LoWPAN is achieved through link-layer forwarding (*Mesh-Under* routing) or IP routing (*Route-Over* routing).

1.3 Protocol stack

In 1.2 the 6LoWPAN protocol stack is shown, which is based on a five-layer OSI model. The most common protocol for the transport layer is UDP but TCP can also be used.

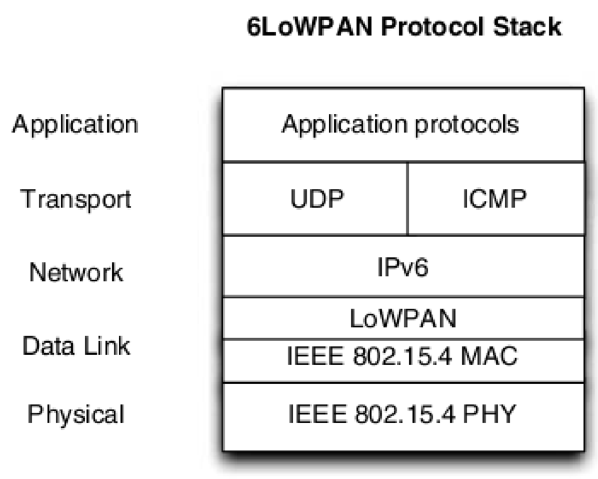


Figure 1.2: 6LoWPAN protocol stack. Source: *6LoWPAN: The Wireless Embedded Internet book[7]*.

Adaptation between the IPv6 header and the 802.15.4 frame is performed by the adaptation layer in edge routers. This transformation is transparent, efficient and stateless in both directions. 6LoWPAN adaptation in an edge router is typically performed as part of the 6LoWPAN network interface driver and is usually transparent to the IPv6 protocol stack itself.

Several link layers can be used in 6LoWPAN:

- **IEEE 802.15.4**. Defines low-power wireless embedded radio communications at 2.4 GHz, 915 MHz and 868 MHz. Provides 20–250 kbit/s data rates depending on the frequency. Channel sharing is achieved using Carrier Sense Multiple Access (CSMA).
- **Sub-GHz ISM band radios**. Sub-GHz ISM bands for unlicensed operation are especially popular in low-power wireless embedded applications such as telemetry, metering and remote control. The sub-GHz ISM bands cover 433 MHz, 868 MHz and 915 MHz. The main reasons

for sub-GHz popularity are the better penetration of lower frequency, resulting in better range compared to 2.4 GHz, and the 2.4 GHz ISM band becoming very crowded in urban environments.

- **Power line communications**. Power line communications (PLC) include home automation, energy efficiency monitoring and smart metering.

1.4 Addressing

IPv6 addresses are 128 bits in length, and consist of a 64-bit prefix part and a 64-bit interface identifier (IID)[9]. 6LoWPAN networks assume that the IID has a direct mapping to the link-layer address. This avoids the need for address resolution. The IPv6 prefix is acquired through Neighbor Discovery Router Advertisement (RA) messages[10] as on a normal IPv6 link. The formation of IPv6 addresses in 6LoWPAN from known prefix information and known link-layer addresses, is what allows a high header compression ratio.

1.5 Forwarding and Routing

Forwarding and routing are used in order to send packets over multiple radio hops. Forwarding and routing can be performed in layer 2 or layer 3.

In case routing and forwarding are performed in layer 2, the routing technique is called *mesh under*. The actual forwarding and routing is not hidden from the 6LoWPAN adaptation layer. As each forwarding step overwrites the link layer destination address by the address of the next hop and the link-layer source address by the address of the node doing forwarding, this information is stored in the *mesh header* introduced by 6LoWPAN.

In case routing and forwarding are performed in layer 3, the routing technique is called *router over*. It does not require any special support from the adaptation layer because before the layer-3 forwarding acts, the adaptation layer has decapsulated the packet.

1.6 Header compression

An important aspect to consider is the limited payload size of packets provided by the IEEE 802.15.4. Most of the payload is consumed by the IPv6 header. 6LoWPAN is more efficient when IPv6 packets fit in a single packet, thus avoiding fragmentation and reassembly. 6LoWPAN employs a *header compression*. There are two types:

- **Stateless header compression**: defines two header compressions to work together: HC1 to compress IPv6 headers and HC2 to compress UDP headers.
- **Context-based header compression**: it is used to compress global addresses. It assumes some additional *context* when joining the 6LoWPAN.

2 Web services

2.1 Introduction to Web services

Although WSN are characterized by having limited resources, it is possible to provide them IP functionality. However, enabling IP in these networks would allow interoperability only at network layer but not at higher layers. Further integration would be possible implementing Web protocols in these networks in order to interact with the existing Web technologies. In fact, interacting with existing Web technologies means using a Web service architectural style.

From a conceptual point of view, "*a service is a software component provided through a network-accessible endpoint. The consumer and the provider use messages to exchange requests and responses abstracting device's resources*"[11]. Essentially, there are two types of Web services to consider: SOAP[12] and REST[13] Web services, each one with its own strengths and weaknesses. In this chapter a comparison between the two is provided and a discussion to decide which one fits best in WSN is provided.

2.2 SOAP Web services

SOAP is an XML[14] language defining a message architecture and message formats. It defines a top-level XML element called envelope, which contains a header and a body (Figure 2.1). The header contains information for routing purposes and Quality of Service (QoS) configuration. The body contains the payload of the message.

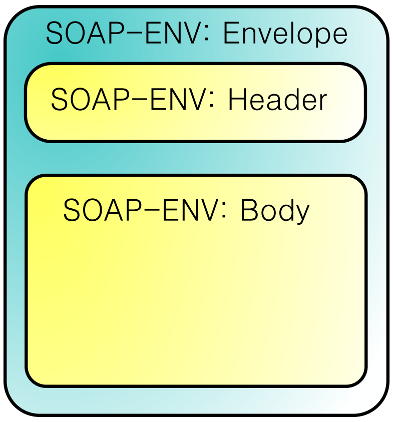


Figure 2.1: SOAP message envelope. Source: *Wikipedia*

SOAP uses Web Services Description Language (WSDL)[15] and Universal Description Discovery and Integration (UDDI)[16] to describe Web services.

WSDL is an XML format for describing network services as a set of end-points operating on messages containing either document-oriented or procedure-oriented information. A WSDL port type contains multiple abstract operations, which are associated with some incoming and outgoing messages. WSDL binding links the set of abstract operations with a concrete transport protocol and serialization formats.

UDDI is a platform-independent, Extensible Markup Language (XML)-based registry for businesses worldwide to list themselves on the Internet and a mechanism to register and locate web service applications

The main advantage of using SOAP is the transparency and independence. A same message can be transported across multiple transport protocols, which can change along the way. QoS aspects can be made independent from the transports used. Using WSDL helps to abstract the underlying communication as well as the service implementation platform. Furthermore, mature SOAP engines and WDSL tools hide the complexity from the application programmer. However, the facility to use SOAP tools can led to some leakage across abstraction levels. Given the lack of standards and rich XML language is very difficult to identify the right construct to express the data.

2.3 RESTful Web services

REpresentational State Transfer (REST)[13] was originally introduced as an architectural style for building large-scale distributed hypermedia systems. It is based in four principles:

- **Resource identification through URI.** RESTful Web services expose a set of resources which identify the targets of interaction. They are identified by URIs.
- **Uniform interface.** There is only a fixed set of operations to interact with a resource: GET, POST, PUT and DELETE. In general only the GET and POST verbs are used.
- **Self-descriptive messages.** Resources are decoupled from their representation. Metadata of the resource is available and used.
- **Stateful interactions through hyperlinks.** Each interaction with a resource is stateless, messages are self-contained.

RESTful Web services are very simple to use. HTTP servers and clients are available for all major operating systems and the default HTTP port (80) is open for most firewall configurations.

To support a larger number of clients there several strategies included in REST such as caching, clustering or load balancing. It can also exchange different messages types from XML such as Javascript Object Notation (JSON).

2.4 SOAP vs REST

From the REST perspective, end-points are seen as a set of resources and the HTTP protocol is seen as an application protocol. To interact with the resources only the four verbs GET, POST, PUT and DELETE must be used. From a compatibility point of view, in the Web, heterogeneity problems relay on Web browsers which can lead to unsupported Javascript libraries or HTML rendering problems. The payload format is negotiated through MIME types, so a REST end-point can serve **responses in different formats**. As resource identification, URIs are used. They are exchanged via URL or hyper links. For the service description, REST uses a more descriptive API form based on informal, textual descriptions.

From the SOAP perspective, end-points are seen as points to exchange XML messages. The HTTP protocol is seen as a transport protocol and the semantics of the message depends on the application. Interoperability is managed with custom XML messages between end-points. The payload format is only one: SOAP. As a resource identification, SOAP allows some kind of addressing information through the definition of end-point references. It uses WSDL for resource description giving an abstract set of operations to interact with an end-point. In this way, incompatibilities between programs can be easily detected because code breaks at compile time. It has a set of optional sets to ensure QoS properties on messages exchanged.

As messages are self-contained in the REST architecture, REST messages produce less overhead than SoAP messages. Also SOAP is much more complex than REST an allows to define a custom semantic for different applications. Limiting semantics like REST does is a better way to achieve interoperability. Also caching techniques are easily applicable to RESTful end-points which is a key feature in WSNs, reducing the number of interactions between nodes.

SOAP also uses verbose formats like XML that makes messaging processing complex and expensive. REST negotiates different formats and uses a more appropriate format depending on the application producing less message overhead.

For reasons mentioned above, REST is a good candidate for WSNs due to its low complexity and fast integration.

3 CoAP application protocol

3.1 Overview

The main objective of the CoAP application protocol is to provide a generic Web protocol for the special requirements of constrained wireless nodes. CoAP is similar to HTTP but its goal is not to simply compress HTTP but to realize a subset of REST operations optimized for M2M interactions. CoAP main features are:

- UDP binding to reduce TCP overhead with optional reliability supporting unicast and multicast requests.
- Asynchronous message exchange.
- Low overhead and parsing complexity.
- New link format.
- URI and Content-Type support.
- Simple proxy and caching capabilities.
- Optional resource discovery and subscription mechanisms.

The interaction model is similar to the client/server model of the HTTP protocol. Clients request an action to a resource. Then, the server sends a response with a response code, which may include a resource representation. Messages are exchanged asynchronously over a datagram-oriented transport. Four types of messages are defined: **Confirmable (CON), Non-Confirmable (NON), Acknowledgment (ACK) and Reset (RST)**.

CoAP follows a **two-layer approach** where a message layer is used to deal with UDP messages and a Request/Response layer to deal with request/response interaction.

3.2 Request / Response model

3.2.1 CoAP messages

As CoAP is bound to the non-reliable protocol UDP, it implements a lightweight reliability mechanism trying to recreate TCP. The main characteristics are:

- Simple stop-and-wait retransmission reliability with exponential back-off.
- Duplicate message detection.
- Multicast support.

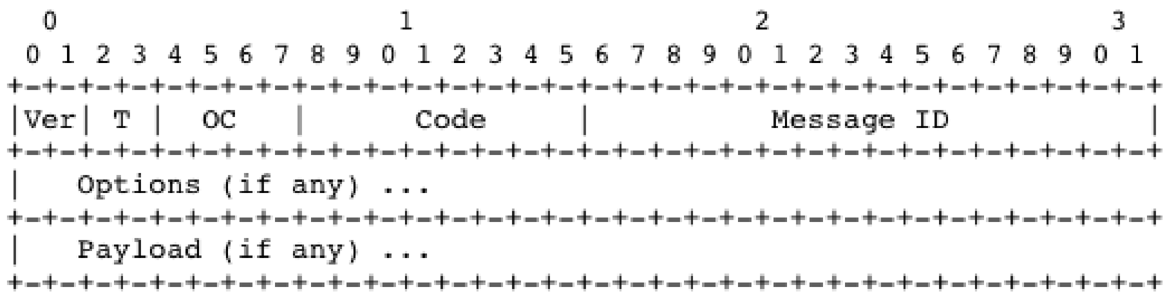
CoAP protocol defines four types of messages:

- **Confirmable (CON):** this message is sent when a reliable transmission is needed. The protocol guarantees that the message will not be lost within certain conditions. Because messages are transported over UDP, the reliability is accomplished with packet retransmission if a response is not received in a given time out. It increases exponentially with every new retransmission and, thus, provides a simple congestion mechanism. The packet will be lost if the maximum number of retransmissions is reached.
- **NON-Confirmable (NON):** this message is sent if a reliable transmission is not needed. It is useful for requests that are sent regularly. This message may carry a response for a NON request.
- **Acknowledge (ACK):** this message carries a response to acknowledge a CON request. This type of messages may carry response data or not. In the first case, the response is called piggy-backed response and in the second case separate response. The second one is used when the server cannot process the request immediately but promises that it will be processed.
- **Reset (RST):** this messages indicates that a CON messages has arrived but there is no context to process it.

3.2.2 CoAP message format

The figure 3.1 shows how a CoAP message. It has three different parts which are transported over an UDP packet:

- **CoAP header**. Provides basic information to recognize the CoAP version, the type of message, a message code and a message identifier. It also provides information to parse the message.
- **CoAP options**. Are used to provide parameters needed to fulfill requests.
- **CoAP payload.** Contains the message body.



```
   0                   1                   2                   3
 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5 6 7 8 9 0 1
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|Ver| T |  OC   |      Code     |          Message ID           |
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Options (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
|   Payload (if any) ...
+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+-+
```

Figure 3.1: CoAP message fields. Source: *CoAP draft version 7.*

The CoAP header has the following fields:

- **Version (Ver)**. Indicates the CoAP version number. Implementations of this specification MUST set this field to 1.
- **Type (T)**. Indicates the message type: CON, NON, ACK or RST.
- **Option Count (OC)**. 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.
- **Code**. Indicates if the message carries a request (code values from 1 to 31) or a response (code values from 64 to 191), or is empty (0). (All other code values are reserved.) In case of a request, the Code field indicates the Request Method; in case of a response a Response Code.
- **Message ID**. Used for the detection of message duplication, and to match messages of type ACK/RST and messages of type CON.

In CoAP, messages can be a request or a response. A CoAP request consists of the method to be applied to the resource, the identifier of the resource, a payload and optional meta-data about the request. There are four basic methods: **GET, POST, PUT and DELETE**.

After receiving a request, a server responds with a CoAP response. There are three types of responses:

- **Piggy-backed**. The response is carried directly in the acknowledgment message. The response is returned in the acknowledgment message independently of whether the response indicates success or failure.
- **Separate**. In some cases, it may not be possible to return a response immediately. In order to avoid packet retransmission, the server sends an ACK to promise the client it will process the request. When the server finally processes it, then a CON message is sent.
- **Non-confirmable**. If the request is not confirmable, then the response is also not confirmable.

A response is identified by the Code field in the CoAP message header. There are three code classes:

- **Success (2.x).** The request was successfully received, understood, and accepted.

- **Client Error (4.x).** The request has bad syntax or cannot be fulfilled.
- **Server Error (5.x).** The server failed to fulfill an apparently valid request.

Response codes are designed to be extensible. If one of them is not recognized, then it must be treated as a being equivalent to the generic Response Code of that class.

3.2.3 CoAP message reliability

A reliable transmission is started marking a packet as confirmable. A recipient must acknowledge such message with an acknowledge message or reject it with a reset message. The sender transmits the CON message at exponential increasing intervals until receives an ACK, RESET or it runs out of attempts. For each time out expired, the time out is doubled.

The recipient should acknowledge each duplicate copy of the CON message using the same ACK but it should process any request or response only once. It should ignore any duplicates and process the message only once.

Unreliable messages are not acknowledged or rejected. If recipient lacks the context to process the message, the message must be simply ignored. The recipient must be prepared to receive the same message multiple times.

3.2.4 Options

Options are identified by an option number. Odd numbers indicate critical options and even numbers elective options. Figure 3.2 shows the option format. Options fields are:

```
  0   1   2   3   4   5   6   7
+---+---+---+---+---+---+---+---+
| Option Delta  |    Length     | for 0..14
+---+---+---+---+---+---+---+---+
|   Option Value ...
+---+---+---+---+---+---+---+---+
                                            for 15..270:
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
| Option Delta  | 1   1   1   1 |          Length - 15          |
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
|   Option Value ...
+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+---+
```

Figure 3.2: Option format fields. Source *CoAP draft version 7*.

- **Option Delta**. 4-bit unsigned integer. Indicates the difference between the option Number of the current option and the option number of the previous option.

- **Length**. 4-bit unsigned integer. Indicates the length of the option Value. When this field is set to 15 an 8-bit unsigned integer is added allowing lengths ranging from 15 to 270 bytes.

Options can be critical or elective. The difference is how an unrecognized option is handled in an end-point:

- **Elective** must ignore messages with unrecognized options.
- **Critical** that occur in a CON message request must cause the return of 4.02 response code.
- **Critical** that occur in a CON message response and in a NON message must silently ignore the message.

There are several types of options:

- **Token**. It is used to match a response with a request. Every request has a client-generated token which the server must echo in any response.
- **Uri-Host**. It specifies the Internet host of the resource being requested. The default value is the IP literal representing the destination IP address.
- **Uri-Port**. It specifies the port number of the resource. The default value is the destination port.
- **Uri-Path**. It specifies one segment of the absolute path to the resource.
- **Uri-Query**. It specifies a query string.
- **Proxy-Uri**. It is used to make a request to a proxy. The proxy is requested to forward the request or service it from a valid cache and return the response.
- **Content-Type**. It indicates the representation format of the message payload given as a numeric value.
- **Accept**. It indicates when included one or more times in a request, one or more media types, each of which is an acceptable media type for the client, in the order of preference.
- **Max-Age**. The maximum time a response may be cached before it must be considered not fresh. When included in a request, it indicates the minimum value for the maximum age of cache response the client will accept.
- **E-Tag**. In a response, provides the current value of the entity-tag for the enclosed representation of the target resource. An entity-tag is intended for use as a resource-local identifier for differentiating between representations of the same resource that vary over time.
- **Location-Path and Location-Query**. It indicates the location of a resource as an absolute path URI. It can be included in a response to indicate the location of a new resource created with POST.

- **If-Match**. It may be used to make a request conditional on the current existence or value of an ETag for one or more representations of the target resource.
- **If-None-Match**. It may be used to make a request conditional on the non-existance of the target resource. If-None-Match is useful for resource creation requests, such as PUT requests, as a means for protecting against accidental overwrites when multiple clients are reacting in parallel on the same resource.

3.2.5 Request / Response Matching

In a piggy-backed response, both the Message ID of the CON message and the ACK MUST match. In a separate response, just the token of the response and original request MUST match.

3.2.6 Caching

Nodes can cache their responses in order to reduce the response time and network bandwidth consumption on future, equivalent requests. The goal of caching is to reuse a prior response message to satisfy a current request. A node must not use a stored response unless:

- The request method and the one used to obtain the stored response must match.
- All options match between those in the presented request and those of the request used to obtain the stored response.
- The stored response is either fresh or successfully validated.

There are two ways to decide if a cache can be used to satisfy a request:

- **Freshness model**. The mechanism for determining freshness is for an origin server to provide an explicit time in the future using Max-Age option. If an origin server wants to prevent caching it must explicitly include a Max-Age option with a value of zero seconds. If the client has certain influence in the freshness calculation it can include a *Max-Age* option in a request.
- **Validation model**. When an end-point has one or more stored responses for a GET request but it can not use any of them, it can use the *E-tag* option in the GET request to give the origin server an opportunity to both select a stored response to be used and to update its freshness. Each stored response has an entity-tag that should be sent to the server via an *E-tag* option. The server response 2.03 (Valid) indicates that the stored response identified by its *E-tag* option can be reused. For any other response, it should be used to satisfy the request.

3.2.7 Proxying

A proxy is a CoAP end-point that can be tasked by clients to perform requests on their behalf. Requests to proxy end-points use CON and NON messages but they specify the request URI in a different way. They use the *Proxy-Uri* option. When a proxy request is made to an end-point that is unwilling or unable to act as a proxy then it must return a 5.05 (Proxy Not Supported) response. If the request to the destination times out, then a 5.04 (Gateway Timeout) response must be returned. If the request cannot be processed, then a 5.02 (Bad Gateway) response must be returned. If a response is generated out of a cache it must be generated with a *Max-age* option.

3.3 CoRE Link Format

Typical use cases for Web Linking on today's web include describing the author of a web page, describing relations between web pages, etc... Web linking can be applied to M2M applications to assist clients in finding and understanding how to use resources on a server. This section gives an overview of all the cases where CoRE Link Format may be applied to solve a need.

- **Resource Discovery:** sometimes there is a need for local clients and servers to find and interact with each other without human intervention. CoRE Link Format can be used in these scenarios in order to find which resources are hosted by a server.
- **Resource collections:** servers can make use of collections of resources. For example a list of alarm nodes or a group of temperature sensors.
- **Resource Directory:** in many scenarios it makes sense to deploy entity servers which store links to resources stored in other servers. For example in constrained networks with sleeping servers. It can be seen as a limited search engine for M2M resources.

3.3.1 Description

Each link conveys one target URI as a URI-reference inside angle brackets. Typically links are used to describe resources hosted by a server. Link CoRE Format can make use of any registered relation parameter or target attributes by including the relation parameter ("rel"). The context of a relation can be defined using the "anchor" parameter. In absence of a relation parameter , a "hosts" relation is assumed and indicates that the target URI is a resource hosted by a server given the base URI or the anchor parameter. Relations between resources hosted on a server, or between hosted resources and external resources can be expressed.

Link extensions can be defined in order to describe useful information in accessing the target link of the relation:

- **Resource type 'rt' attribute:** Noun describing the resource.
- **Interface description 'if' attribute:** Verbs describing the resource. Multiple interface links may appear in a link.
- **Content-type code 'ct' attribute:** Internet media type this resource returns. Alternatively, the "type" attribute may be used to indicate an Internet media type as a quoted-string.

- **Maximum size estimate 'sz' attribute:** used to know an indication of the maximum size of the resource.

3.3.2 Resource discovery

Resource discovery is accomplished through the use of a known resource URI which returns a list of links about resources hosted by that server and other link relations. Well-known resources have a path component that begins with "/.well-known/". For CoRE Resource Discovery the "./well-known/core" resource is defined. It supports the following interactions:

- Performing a GET to the default port returns a set of links available from the server in the CoRE Link Format.
- Filtering may be performed on any of the link format attributes using a query string. However, the server is no required to support filtering.
- More capable servers could support a resource directory by requesting the resource descriptions of other end-points or allowing to POST requests to "./well-known/core".

An example of resources presented by the server is the following:

```
REQ: GET /.well-known/core
RES: 200 OK
</sensors>;ct=40;rt="index";rt="Sensor Index", </sensors/temp>;rt="TemperatureC";if="sensor",
</sensors/light>;ct=41;rt="LightLux";if="sensor",
<http://www.example.com/sensors/t123>;anchor="/sensors/temp"
;rel="describedby",
</t>;anchor="/sensors/temp";rel="alternate"
```

In this particular example, the server presents a directory called *sensors* that contains two resources: *temp* and *light*, available through the URLs */sensors/temp* and */sensors/light* respectively . The attributes included in each resource, describe the resource. The meaning of these fields was explained in 3.3.1.

3.4 Observing resources

The state of a CoAP server can change over time. Polling or long-polls generate significant complexity and overhead. A much simpler mechanism is provided to solve the basic problem of resource observation. Observing CoAP resources is implemented following a subject/observer design pattern where an object called subject maintains a list of interested parties called observers and notifies them automatically when a predefined condition , event or state change occurs.

3.4.1 Phases

- **Establishment** A client registers itself with a resource by performing a GET request that includes the *Observe* option. When the server receives such a request, it services it like a GET request without this option and if response indicates success, then establishes a relationship between the client and the target resource. The token specified by the client will be echoed in all notifications sent and will also include an *Observe* option to indicate that that the observation relationship was successfully established. If the server is unable or unwilling to establish observation relationship it must silently ignore the *Observe* option and process the request as usual.

- **Maintenance** A client may refresh an observation relationship at any time. It is recommended that the client does not refresh the relationship for the time specified in the *Max-Age* option. A client refreshes an observation simply repeating the GET request with the *Observe* option. When the server receives such a request, it must replace or update the existing one. A request relates to the same observation relationship if:

  - The URI of the two requests match.
  - The source of the two requests match,
  - The message Ids and any *Token* options in the two requests must not be taken into account.

- **Termination** The observation relationship ends when one of the following conditions occurs:

  - The server sends a notification response with an error code (4.xx or 5.xx)
  - The client rejects a CON notification with a RST message.
  - The last attempt of transmitting a CON notification to the client times out.

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

  - Rejects a CON notification with a RST message.
  - Performs a GET request on the resource without an *Observe* Option.

3.4.2 Notifications

- Each notification response must include an *Observe* option and echo the token specified by the client in the request.

- A notification should have a 2.05 (Content) or 2.03 (Valid) response code.

- The representation format must be same than the original GET request. If the server is not able to continue sending notifications in such format, it should send a 5.00 (Internal Server Error) response.

- Notifications can be sent CON or NON messages.

- If the client does not recognize the token in a CON notification message it must not acknowledge the message and should reject it with a RST message.

- Notifications may arrive in a different order than sent by the server. If the notification arrives before the response to the initial request, the client can take the notification as initial response in place of actual response.
- Notifications may be cached by CoAP end-points under the same conditions as with all responses.
- If a server is waiting for an acknowledgment from the client and a new notification message is prepared to be sent, it must stop retransmitting the old notification and attempt to transmit the new one.

3.4.3 Reordering

Notifications may arrive in different order and the main objective for the client is to maintain a eventual consistency. For this purpose, the server keeps a single 16-bit unsigned integer variable. This variable is incremented approximately every second. The server must include the current value of that variable as the value of the *Observe* option each time it sends a notification. In order to discard not valid notifications, the procedure explained in [30] can be followed.

4 TinyOS operating system

4.1 Overview

As defined in [3]: "*TinyOS is an open source, BSD-licensed operating system designed for low-power wireless devices, such as those used in sensor networks, ubiquitous computing, personal area networks, smart buildings, and smart meters. A worldwide community from academia and industry use, develop, and support the operating system as well as its associated tools, averaging 35,000 downloads a year*".

TinyOS makes building network applications easier. Its main characteristics are:

- **A component model**. TinyOS is written in nesC language which allows the developer to create small pieces of code and explicitly declare its dependencies.
- **A concurrent execution model**. Allows the developer to create tasks and control the scheduler preemption. This supports many components needing to act at the same time while requiring little RAM.

  In TinyOS each I/O call is split-phase: rather than block until completion, a request returns immediately and the caller gets a callback when the I/O operation is complete. Since the stack is not tied up waiting for I/O calls to complete, TinyOS only needs one stack and does not provide threads.

  To defer a procedure call, TinyOS uses tasks. Any component can post a task that TinyOS will run at some point later. Because nodes must spend the most of their time asleep, tasks tend to run very soon after they are posted. Although tasks cannot preempt each other (no concurrency problems appear between tasks), low-level interrupts can preempt a task. TinyOS provides a mechanism to deal with possible race conditions[17].
- **Application programming interfaces, services and component libraries**. Simplifies writing new applications and services.

4.2 Memory allocation

A very valuable resource in motes is RAM. Not only because it is limited but causes power consumption when the mote is asleep. TinyOS modules allocate memory by declaring variables which are private to the component. Because TinyOS uses split-phase operations and does not provides threads, there is no long-lived stack-allocated data.

Modules often need constants. In order to save RAM constants should be defined as C `enum` types. Then, they will not be stored on RAM. It is better than the `define` directive because it exists in the debugging symbol table and application metadata. As `enum` types can only declare integers, the `define` directive can be used for float-point and string constants.

Is not a good practice the use of dynamic memory allocation through `malloc` calls or other C libraries. The lack of memory protection on most embedded microcontrollers makes their use particularly risky.

If the heap grows up and the stack grows down since there is no hardware memory protection the two can collide. Instead, it is convenient to allocate memory as module variables. In some cases it can be a solution to use a shared memory pool with allocated memory. When the program needs some memory, it can request it to the memory pool and return it back when is no longer necessary. In all the cases, by avoiding the use of the heap, the only run-time failure point is the stack. To avoid stack overflow, the developer should avoid recursion and not declare any large local variables.

4.3 Memory ownership and split-phase calls

TinyOS programs contain many concurrent activities such as receiving radio packets, sending packets, processing packets, etc... Ensuring that these activities do not step on each other by accessing each other's data is often a complex problem.

The only way components can interact is with function calls. Each function call is called **command** in TinyOS. When some operation is complete, a callback function is called. Each callback function is called **event**.

There are two ways to pass parameters to a command: by value or by reference. In the first case, the value is copied into the stack, so the callee can freely modify it. In the second case, the caller and the callee share a pointer to the data memory region. Thus, the two components need to manage access to this memory area in order to avoid memory corruption. If the caller uses the memory passed to the callee at some point between the calling and the callback, memory corruption happens because the memory is shared between the two.

To avoid these problems, TinyOS follows an ownership discipline: at any point, every memory object should be owned by a single module. A command is said to pass ownership when it passes the ownership from caller to callee. Then, the callback should return the memory object to its original owner (the caller).

A common situation found in most of the applications is packet reception. Each packet reception needs a buffer to copy the data. There are two main options used by TinyOS programmers:

- **The reception callback has a parameter that contains a pointer to the buffer with the packet**. In this case, to use the data in some point of the program, it should be copied to a new data buffer. When the callback finishes, a new callback can be executed and the pointer given by the old callback will be reused. This can cause memory corruption problems if the program uses this memory at some point.
- **Using the buffer-swap strategy.** It happens with events defined like this example:

  ```
  message _ t * Receive.receive(message_t * msg , void * payload , uint8_t len)
  ```

  In this case the event also provides a pointer to the buffer data but this pointer can be reused somewhere else in the application. This happens because this event returns a pointer to a memory buffer that will be used in next reception. In order to make this work, upper layers should keep a free memory list. When a new packet is received, a new pointer from the free memory list will be returned for next reception. Then the pointer given by the event has no memory corruption problems. An application implementation can decide to return the same pointer that the event provides.

4.4 Execution model

As seen in previous sections, all operations in TinyOS are split-phase. In place of threads, all code in TinyOS is executed either by a task or an interrupt handler. A task is nothing more than a deferred procedure call that will be executed at some point by the TinyOS scheduler and runs to completion. This means that if the scheduler decides to execute a certain task, no other tasks can preempt the current task until it finishes. Interrupts, in contrast, can occur at any time, interrupting tasks or other interrupt handlers.

Because code executed from an interrupt handler must be aware of concurrency problems, nesC distinguishes between **synchronous** code (sync) that can only be executed from a task and **asynchronous** (async) that may be executed from both tasks and interrupt handlers. Asynchronous code must be aware of race conditions and that is why nesC provides advanced tools to deal with these problems.

5 CoAP Web Services integration and discussion

5.1 Overview

In this section, a solution to integrate Internet applications with WSNs based on CoAP is discussed. In the Web world, HTTP protocol is the most popular protocol for the application layer and uses TCP in the transport layer. In contrast, WSNs nodes can use CoAP in the application layer and UDP in the transport layer. In order to make them compatible, some kind of translation must be made.

CoAP application protocol is very similar to HTTP in its basic functionality. On one hand, they both provide an interface for RESTful applications. On the other hand, they both provide a reliable transmission: HTTP with a TCP connection and CoAP with its own retransmission mechanism. Seeing that they are conceptually very similar, it makes sense to translate between CoAP and HTTP.

The following sections describe a solution for HTTP-CoAP integration. First, a network scenario is proposed and the gateway requirements are described. Next, possible alternatives are described in order to fulfill the requirements.

5.2 Requirements

The network scenario proposed is represented in figure 5.3. The first requisite is to be able to access external IP networks. An edge router is used for this purpose. In order to allow HTTP clients to access CoAP servers, the same edge router can be employed.

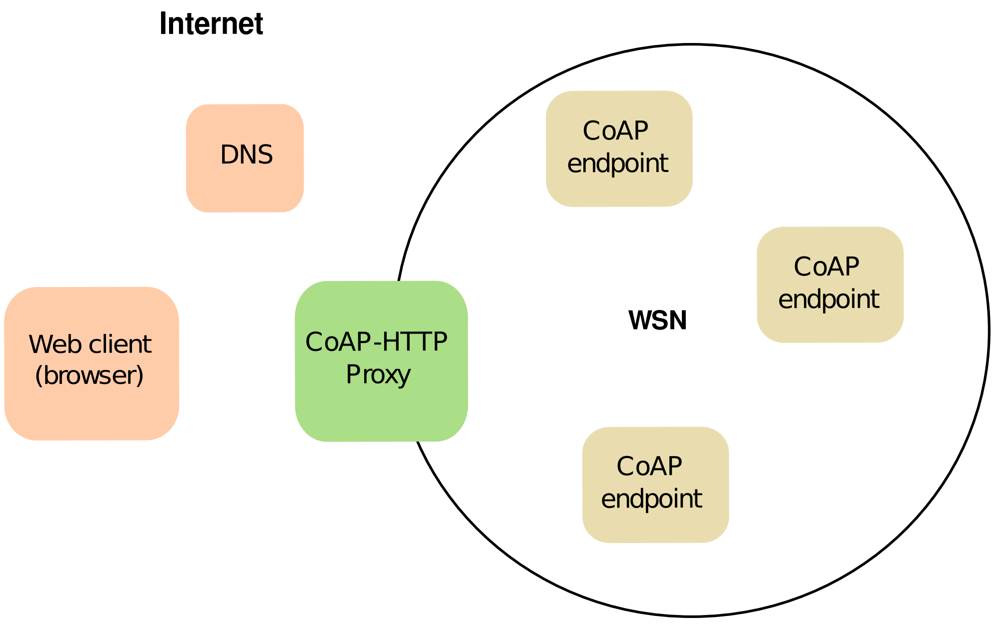


Figure 5.3: Network scenario

In general, in order to maximize the WSN throughput a CoAP proxy is used. The proxy receives CoAP requests and creates new requests to contact other servers. Usually the main advantage of a proxy is the response caching, storing responses that were previously sent. This proxy can be embedded in wireless node or in a more powerful device such an edge router.

From an implementation point of view, there are some problems that wireless nodes can have. On one hand, the proxy must maintain all requests in a queue until a response is received or a time out expired. This means that these requests consume RAM memory. On the other hand, implementing a cache system means that also responses must be kept in node's memory. Due to the memory and energy limitations of these devices, deploying a CoAP proxy in a wireless node is impractical.

As a conclusion, a CoAP proxy can be implemented in a wireless node but the energy consumed would increase (and therefore battery life would be dramatically reduced) and the number of clients allowed would be very limited due to the limited RAM memory. **As a result proxying is usually performed by a more capable embedded device**.

Using a capable embedded device can offer extra services to the network. **In this work, the following services have been taken into consideration**:

- **Accessing external IP networks**. As mentioned before, the new device must act as an edge router to allow CoAP end-points access other networks and make the network visible for other end-points. Thanks to this, devices connected to the IP network can connect to CoAP end-points using an IPv6 address and each CoAP endpoint can connect to the IP network.

- **CoAP-HTTP proxying**: provides the ability to communicate HTTP clients with CoAP end-points. As a consequence, a translation between CoAP and HTTP is needed at this point. It is also interesting if CoAP end-points can also send CoAP requests to HTTP servers.

- **Caching**: Caching is essential in a WSNs. It is supported by CoAP protocol using a *freshness* or *validation* model. It can be implemented using a persistence layer to store data so that when the device is restarted the data is still existing or storing the cached responses in RAM. In this work we propose the first option which makes the device more robust to possible problems.

- **CoAP proxying**: CoAP end-points can make CoAP requests through the network device if it implements request proxying. Proxying is supported by the CoAP protocol using the PROXY_URI option. This service is usually combined with caching allowing better network throughtput.

- **Service discovery**: One problem in machine-to-machine interactions is how all devices are concerned about the presence of other end-points with different services. This is important because the communication between end-points should be machine-to-machine mainly and should be able to work without human intervention. In [18] a service discovery for new devices is proposed. The **./well-known/core** PATH in a server is employed to get a list of its resources in the CoRE link format. The new device can serve as a small search engine allowing CoAP end-points to discover other end-points with similar or different capabilities. For instance one sensor may want to know which end-points are capable of measuring the environment temperature.

From now on, this new device will be referred as **CoAP proxy**.

The discussion now focuses on technical decisions that must be made to get all these set of services. For the CoAP-HTTP service, a mechanism for handling HTTP requests must be analyzed. The first immediate solution is to create the HTTP handling in some programming language from scratch. This solution implies a lot of effort. Moreover, all underneath problems have been addressed by many people before and there are good open source Web servers and clients that handle the HTTP protocol.

Typically HTTP requests are handled by Web servers. A possible solution could be using a Web server to handle HTTP requests and then offer all the desired services. Then, the Web server must allow HTTP request and response handling customization. There are two options: creating a Web server module which basically handles the HTTP request or using an external program that communicates with the Web server and returns the HTTP response based on the request. In both cases the HTTP handling is performed by the Web server. Available options use Web servers for HTTP request handling are discussed and analyzed in next section.

For caching capabilities one decision should be made: using RAM to store CoAP responses temporary or using a persistence layer and store requests in the device file system. In this work the second option is chosen because provides better robustness.

It is also important to consider how the device will be maintained. In this work, a Web configuration and management tool is proposed. Therefore, the device can be configured and rebooted from external Web browsers through this Web interface.

5.3 Web server integration

In section 5.2 requirements for integrating existing Web technologies with WSNs were discussed. One of the most important decisions made was using an existing Web server to handle the incoming requests. In this section different possibilities to communicate the Web server with external applications are reviewed.

5.3.1 CGI

As described in [19]: *"The Common Gateway Interface (CGI) is a standard that defines how Web server software can delegate the generation of Web pages to a stand-alone application or an executable file. Such applications, known as CGI scripts, can be written in any programming language, although scripting languages are often used"*.

Basically CGI allows external applications (CGI scripts) to process a HTTP request, generate a response and send it back to the Web server. This allows the creation of dynamic Web sites where the served content is no static and changes between requests.

When using CGI, each time the Web server receives a new request to the CGI application:

1. Creates a new process.
2. Executes the CGI script in the previously created process and sends request parameters through environment variables to a CGI script.
3. The CGI script receives the HTTP request, makes some logic and finally sends back a response to the Web server using the standard output. The response includes HTTP headers and must return the Content-Type header.
4. When the CGI script finishes sending the response, the server removes the process.

Advantages:

- Process isolation. As the CGI script runs in a separate process, if some problem occurs it does not affect the Web server.
- Web server independence. Applications are portable between different Web servers.

Drawbacks:

- Process overhead creation.
- Not suited for long-lived applications. The CGI script is executed only when a new request arrives and is only running during the request life cycle.

5.3.2 Web server Module

Another option is create a Web server module. This can be seen as a Web server extension of the request handling. This functionality is extended using different tools that the Web server provides basically to manipulate HTTP messages and send responses back. It is conceptually similar to the CGI option.

Advantages:

- There is no process overhead. A module runs inside the Web server because is an extension.

Drawbacks:

- Web server dependent. The developer needs to write a custom module for each different Web server used.
- Not suited for long-lived applications. A module code is executed only during the request life cycle.

5.3.3 FastCGI

FastCGI[20] provides an alternative to the "one new process per request" paradigm that CGI proposes.

Advantages: It solves the following issues:

- No new process per request. Request environment information is sent to a fastCGI daemon that sends back a response.
- Suited for long-lived applications. There is a fastCGI daemon running pending to serve new requests.

Drawbacks:

- It is usually more difficult to program a FastCGI daemon that implementing a CGI script.

5.3.4 Conclusions

The following table summarizes the alternatives to integrate external applications with a Web server. Red cells are not desirable and green ones desirable.

Alternative	Web server compatible	Process isolation	Process overhead	Long-lived	Complexity
CGI	Yes	Yes	Yes	No	Low
Module	No	No	No	No	High
FastCGI	Yes	Yes	No	Yes	High

Table 1: Web server integration possibilities

The choice made in this work is FastCGI. As can be observed, FastCGI is the only one suited for long-lived applications. This feature is important to handle subscriptions. With FastCGI a daemon can be pending of receiving subscription notifications. Although it is also possible using other alternatives, a daemon must be created and communicated with CGI or module. In all cases, some kind of protocol is needed for the interconnection between the daemon and the custom program. FastCGI provides this feature directly avoiding the "one process per request" paradigm. The complexity can be sorted using good libraries available for multiple programming languages, including C [1].

5.4 Subscriptions

A subscription is the ability to notify that an event has occurred in a CoAP end-point. The objective of this subsection is to review different options available to integrate subscriptions with Web servers and Web browsers. As CoAP provides a simple subscription mechanism, the proxy must convert a CoAP subscription into an "HTTP subscription". To handle a "HTTP subscription" there are two main possible options:

- CoAP end-points may send a POST request to a Web server in order to log an event, for instance storing received parameters in a database. However, it is not suitable for real time communication with a Web browser because it does not know when a response arrives unless it makes a HTTP request to the server an gets a list of the stored parameters.

- Web browser real time application. The browser receives the information in real time. There are different ways to enable real time communication with the Web browser. Some of them simulates a real time communication polling the Web server but there also ways to enable efficient bidirectional communication with the Web browser. In this work three real time techniques are studied: polling, long polling and WebSockets.

5.4.1 Polling

Polling is a technique where the Web browser sends requests to the Web server periodically. As the Web is request-response based, the browser needs to request the server for new data available. This method wastes bandwidth and it can have high latency: it can happen that a lot of requests are sent if there are no changes for a long time and if the response is available maybe it takes time to arrive because there is no polling in that moment.

[1]Visit http://www.fastcgi.com for more information. Good documentation and examples are provided.

5.4.2 Long polling (Comet programming)

Generally the Web browser establishes a persistent TCP connection with the server and it does not terminate when the response is sent. Prior to persistent connections, a new TCP connection was established by each request[21]. Some benefits of persistent connections are:

- CPU saved in routers and hosts.
- The browser can make multiple requests without waiting for a response.
- Network congestion is reduced because there are less TCP opened connections.
- Latency is reduced in subsequent requests.

Thanks to persistent connections more efficient real time applications can be developed and some techniques appeared. Comet programming[22] can be defined as any technique that uses a long-lived HTTP connection to reduce the latency which messages are passed to the server.

One of the most widely-used techniques today to enable Comet applications in Web browsers is the so-called Long Polling[23]. It works similarly to simple polling. The client sends a request and the server instead of sending a response if there is no data available it holds until the data is available. If new data is available, it is immediately sent to the client and low response latency is accomplished. Once the response is received, the client establishes a new connection to the server. The HTTP request overhead is still existing but the latency is reduced.

5.4.3 WebSockets

The WebSocket protocol[24] enables a two-way communication between the Web browser and the Web server. The protocol is very simple conceptually: the Web browser is able to receive data from the server without requesting a response. The most interesting part is that there is no HTTP overhead to send notifications.

The communication has two phases:

1. **The handshake**. The client sends a special request to the server to establish a channel and start communicating.
2. **Data transfer**. Once the handshake is done, the Web browser and the Web server can exchange messages without using HTTP requests.

To start a handshake, a HTTP request is sent by the Web client:

```
GET /demo HTTP/1.1\r\n
Host: example.com\r\n
```

```
Connection: Upgrade\r\n
Sec-WebSocket-Key2: 12998 5 Y3 1 .P00\r\n
Sec-WebSocket-Protocol: sample\r\n
Upgrade: WebSocket\r\n
Sec-WebSocket-Key1: 4 @1 46546xW%0l 1 5\r\n
Origin: http://example.com\r\n
^n:ds[4U\r\n
```

In this particular example, the resource **demo** is requested on the **example.com** Web server. To indicate that this request is a handshake, the *Connection: Upgrade* and *Upgrade: WebSocket* headers are sent. To establish a handshake, extra information is needed: *Sec-WebSocket-Key1*, *Sec-WebSocket-Key2*, *Origin* headers and the request body. The server follows the following process with this data:

1. With *Sec-WebSocket-Key1*, it counts how many spaces this string has (**count1**) and creates number (**number1**) taking the string digits.
2. With *Sec-WebSocket-Key2*, it counts how many spaces this string has (**count2**) and creates number (**number2**) taking the string digits.
3. It creates two strings in big-endian format diving the numbers obtained by the number of spaces found: **string1** = **number1/count1** and **string2** = **number2/count2**
4. It joins these two obtained strings with the request body. The MD5 sum of this new string (**string3**) will be sent as a response body.

The server response for the previous case is:

```
HTTP/1.1 101 WebSocket Protocol Handshake\r\n
Upgrade: WebSocket\r\n
Connection: Upgrade\r\n
Sec-WebSocket-Origin: http://example.com\r\n
Sec-WebSocket-Location: ws://example.com/demo\r\n
Sec-WebSocket-Protocol: sample\r\n
\r\n
8jKS'y:G*Co,Wxa-
```

As can be observed, the *Origin* header must be returned as the *Sec-WebSocket-Origin* header and the WebSocket URL location as *Sec-WebSocket-Location.* As explained before, the response body is **string3** and if correct, it indicates that the handshake is complete.

Once the handshake is complete, messages are exchanged starting with the character **0x00** and finishing with **0xFF**.

5.5 Caching

The goal of caching is to eliminate the need to send requests in many cases and to eliminate the need to send full responses in many other cases. HTTP and CoAP provide cache capabilities based on an **Expiration model** and a **Validation model**. In CoAP, cache options are controlled via CoAP options (*Max-Age* and *E-Tag*). In HTTP the *Cache-Control* header provides explicit directives to HTTP caches and it is also using other headers like *Expires* or *E-Tag*. In this section an explanation about cache possibilities in both protocols is provided.

5.5.1 Expiration model

CoAP application protocol provides a simple expiration model called **Freshness model**. Responses include the *Max-Age* option which value indicates how many seconds the response will be considered fresh (valid). With this mechanism, there is no need to request a CoAP server and it really avoids a full response sending.

HTTP provides a similar expiration model[21]. The server specifies an expiration time either with the *Expires* header, or with the *max-age* directive of the *Cache-Control* header. The *Max-Age* directive takes priority over *Expires* header. This means that if the *Max-Age* is present the freshness lifetime will be its value. Otherwise, the freshness value will be the *Expires* header.

5.5.2 Validation model

When a cache is found to use a stored response, first this cache has to be validated with the origin server. If the origin server (or a possible proxy) decides that the cache is still valid, it sends a partial response which indicates that the cache is still fresh.

In CoAP, the validation model is accomplished using the *E-Tag* option in a GET request. This gives the origin server an opportunity to update cache's freshness. If the cache is still valid, it sends a 2.03 (Valid) response that means that the cache identified by the *E-Tag* included in the *E-Tag* option of the response is still valid.

In HTTP, the response sent by a server can attach some sort of validator. When the client wants to know if a cache is fresh, it needs to ask the server to check the validator sent against the current validator and, if they match, a 304 (Not Modified) response is sent. If not, a full response is sent. There are several validators available in HTTP:

- **Last-Modified Dates**: The cache is considered fresh if the entity has not been modified since the *Last-Modified* value.
- **Entity Tag Cache Validators**. The *E-Tag* response-header field value, an entity tag, provides for an "opaque" cache validator. There are two types of entity tags: strong and weak. A strong entity changes if the resource changes but a weak entity may not change. A weak entity can be used to give a semantic to resource changes.

The preferred behavior for a HTTP origin server is to send a *strong entity tag* and a *Last-Modified* header. If it is not possible to use a strong entity tag, then a weak entity tag can be used. The *Last-Modified* header can be elided if there is a risk of a break in semantic transparency.

6 CoAP protocol implementation on TinyOS

In this section, a software solution to integrate the CoAP application protocol into the TinyOS operating system is proposed.

6.1 Hardware

Nodes used in this work should be wireless, small and low-cost and operate unattended for long periods. This sensor nodes, commonly referred as *motes* tend to have very limited computational and communication resources. In order to use a development platform to test the software created, a TelosB platform has been chosen. In figure 6.1 a picture of the hardware platform used can be found.

TelosB was published and developed to the research community by UC Berkeley. As defined in [5]: "*Crossbow's TelosB mote (TPR2400) is an open source platform designed to enable cutting-edge experimentation for the research community. The TPR2400 bundles all the essentials for lab studies into a single platform including: USB programming capability, an IEEE 802.15.4 radio with integrated antenna, a low-power MCU with extended memory and an optional sensor suite (TPR2420)*".
Some hardware characteristics are:

- IEEE 802.15.4/ZigBee compliant RF transceiver.
- ISM band: 2.4 to 2.4935 GHz.
- 250 Kbps rate.
- Integrated onboard antenna.
- 8 MHz MSP430 microcontroller with 10 KB of RAM.
- Low current consumption.
- 1MB external flash for data logging.
- Programming and data collection via USB.

This hardware works with the TinyOS operating system which provides a programming framework and a ready-to-use protocol stack.

6.2 Software

In order to implement the CoAP protocol in TinyOS some software is needed. First a brief analysis of an existing alternative for TinyOS is presented and discussed. Next, a new CoAP library for TinyOS is presented.

Figure 6.1: TelosB hardware

6.2.1 Existing software

An existing implementation of CoAP on TinyOS is available at [25]. It implements *draft-ietf-core-coap-03*, *draft-ietf-core-observe-00* drafts. It is based on the libcoap C library. This implementation has several problems.

At the time of this writing, CoAP CoRE draft has been updated to version 7 but this implementation provides version 3. Also CoAP observe draft has been also updated to version 2.

The libcoap C library reserves memory using `malloc` calls. This is generally a bad practice in TinyOS programing because some devices lack of *memory management unit (MMU)*. It is recommended using statically-allocated elements. In order to simulate dynamic memory allocation, TinyOS *PoolC* component can be used.

CoAP PDU C structure is defined as follows in the libcoap library:

```
typedef struct {
    coap_hdr_t *hdr;
    unsigned short length;
    coap_list_t *options;
    unsigned char *data;
} coap_pdu_t;
```

This message structure maintains a pointer to a data buffer that contains the CoAP packet. Fields `hdr` and `options` are pointers to some positions in the data buffer. It also provides some functions to add options and add payload data. On one hand, if options and payload are added to the CoAP message and a new option wants be added later, payload memory region must be moved so that the desired option can fit into the packet. On the other hand, if memory must be statically allocated, this means that for each PDU used in a component a buffer with the maximum CoAP message length must be allocated. If the packet maximum length is big enough, and the device is very constrained (i.e 10Kbytes of RAM) most of the memory is used only because of the packet buffer.

For the reasons mentioned above, a new CoAP library is developed using the TinyOS component model and allocating memory buffers statically.

6.2.2 CoAP library and components overview

In this subsection a new CoAP library for TinyOS that uses the BLIP TinyOS stack[26] is presented. The library follows a two-layer approach. One layer deals with CoAP packet sending and reception and the other with CoAP request/response interaction.

The design requirements for the CoAP library are:

- Use the less RAM possible.
- Simple design possible.

On one hand, if the program running in the mote uses less RAM memory, it presents less current consumption. This fact improves battery life. On the other hand, TelosB hardware has only 10 KB of RAM memory, so it is a very valuable resource. Second requirement is important in order to maintain and support the software and avoid unnecessary bugs.

CoAP packet reception and sending is accomplished using BLIP UDP components and several new TinyOS components, configurations and interfaces to create CoAP structures from the received UDP packets:

- **UDP packet reception and sending**: uses the *UdpSocketC* generic configuration from BLIP library. It provides the *UDP* interface that can send and receive UDP packets using the BLIP 6LoWPAN stack.
- **CoAP packet creation**: *CoapPduC* generic configuration provides commands to create and manipulate CoAP packets. They use *CoapListC* generic configuration for the internal linked lists and *CoapOptionC* generic configuration for CoAP option creation and manipulation. These three generic configurations provide *CoapPdu*, *CoapList* and *CoapOption* interfaces.

CoAP request/response interaction is based on two TinyOS interfaces: *CoapServer* and *CoapClient*. These two interfaces provide access to several commands and events to send and receive CoAP packets.

In appendix A a detailed explanation of used components and interfaces can be found.

6.2.3 CoAP packet reception and sending layer

As mentioned before, CoAP components are built on top of the TinyOS BLIP library. It provides the necessary TinyOS interfaces to use IP and transport layer protocols such as TCP and UDP. As CoAP uses UDP, only UDP interfaces are used.

BLIP stack provides the *UdpSocketC* generic configuration that provides the interface *UDP* with the following TinyOS commands and events:

- **command error_t bind(uint16_t port)**: binds a local address on the port specified by *port*.
- **command error_t sendto(struct sockaddr_in6 *dest, void *payload, uint16_t len)**: sends an UDP packet to the destination address *dest*, with payload contained in buffer pointed by *payload* which length is *len*.
- **event void recvfrom(struct sockaddr_in6 *src, void *payload, uint16_t len, struct ip_metadata *meta)**: receives an UDP packet from address *src* and places the received packet in the buffer pointed by *payload*. Provides also the length of the received packet *len* and ip metadata *meta*.

The `recvfrom` event is synchronous and only provides a pointer to a buffer with the packet received. When TinyOS signals the `recvfrom` event, there will be other tasks running. In order to work properly, operations executed in the `recvfrom` event must be as short as possible. Note that the `recvfrom` event provides a pointer to a buffer that will be reused in next reception. Thus, one operation needed at this point is saving the UDP packet somewhere else.

A solution is to save the UDP packet and post a task to process the UDP packet. However, if packet reception is faster than packet processing, at some point of the execution more memory will be needed. Memory is allocated statically in components, so if the program does not finally need such amount of memory, it will incur in current consumption. For this reason, there is a trade off between the maximum number of packets that the program is able to receive and process, and memory requirements.

CoAP TinyOS library provides the *CoapPdu* interface to create and manipulate CoAP packets. This interface is used when a new UDP packet is received or when a new CoAP packet is sent. Basically, this interface is able to allocate new `coap_pdu_t` structures using a `PoolP` component. A `coap_pdu_t` structure is defined as follows:

```
typedef struct {
    uint8_t timestamp;
    coap_hdr_t hdr;
    struct sockaddr_in6 addr;
    uint8_t payload[MAX_PAYLOAD];
    uint16_t payload_len;
    coap_list_t opt_list;
} coap_pdu_t;
```

timestamp field is used to know when the CoAP message was created or received. *addr* contains the destination address to send the CoAP message or the origin address from which the packet was received. *payload* is a buffer containing the CoAP message packet received or to be sent and *payload_len* contains the actual packet length. As memory should be statically-allocated the payload buffer size must be the maximum payload length that the mote is able to receive and send.

`coap_hdr_t` contains the CoAP message header defined in 3.2.2 and is defined as follows:

```
typedef struct {
    uint8_t version;
    uint8_t type;
    uint8_t code;
    uint8_t optcnt;
    uint16_t id;
} coap_hdr_t;
```

`opt_list_t` contains the list of options that are present in a CoAP message. `coap_list_t` structure models a generic linked list and is defined as follows:

```
struct coap_node_t {
    struct coap_node_t *next;
    uint16_t key;
    void *data;
};

struct coap_list_index_t {
    struct coap_list_t *list;
    struct coap_node_t *this, *next;
};

struct coap_list_t {
    struct coap_node_t *first;
    struct coap_list_index_t  iterator;
};
```

`coap_list_t` is a linked list that is a data structure consisting of a group of nodes which together represent a sequence. In this case it contains `coap_node_t` elements. Figure 6.2 shows the linked list proposed for the CoAP TinyOS library. Each `coap_node_t` contains a unsigned integer key (*key*) that identifies the *data* pointer which points to region in memory that contains the actual data. This linked list is intended to be general but in this case it is used to store pointers to `coap_option_t` elements. This linked list also contains an iterator (*iterator*) that is used to easily loop through the linked list.

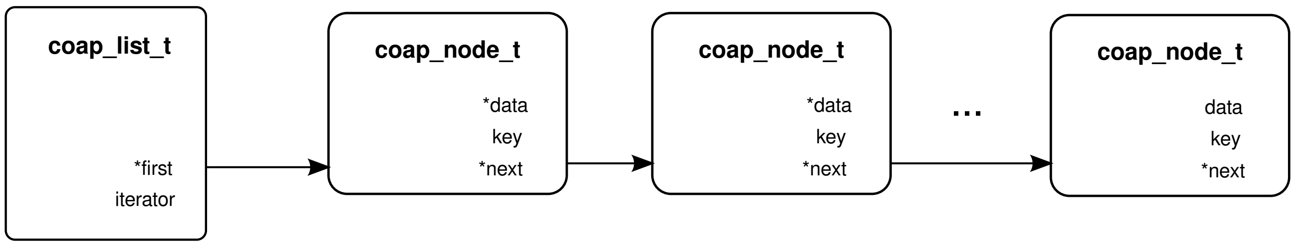


Figure 6.2: Linked list proposed for the CoAP TinyOS library.

A `coap_option_t` structure is defined as follows:

```
typedef struct {
    uint8_t code;
    uint16_t len;
    uint8_t data[MAX_OPT_DATA];
} coap_option_t;
```

Again, as memory should be statically-allocated CoAP option data buffer size must be the maximum possible length an option can have. The actual option length is stored in *len* and *code* contains the CoAP option code. Possible option codes can be found in appendix C.5.

To handle `coap_pdu_t`, `coap_option_t` and `coap_list_t` structures several TinyOS interfaces are proposed:

- **CoapPdu**: allocates new `coap_pdu_t` structures and allows to get and set CoAP options.
- **CoapList**: allocates new `coap_list_t` structures and allows to loop through the linked list and to add and remove list elements.
- **CoapOption**: allocates new `coap_option_t` structures.

Memory allocation uses the TinyOS *Pool* interface that allows to specify a memory region where to get memory. Each new configuration *PoolC* has a parameter specifying which type of structures the pool returns and the maximum number of elements that can be returned. Once the memory is not needed, it must be returned to the pool for later use. With TinyOS pools the same effect than dynamic memory allocation is accomplished.

All three interfaces mentioned before contain a TinyOS *PoolC* component and a command to get the desired structure from the pool. The interfaces provided act as wrapper for TinyOS memory pools.

The reason to work in this way is to separate the buffer needed to contain the CoAP packet to be sent and the actual memory used by each PDU. Programs will contain a buffer with the maximum length the CoAP message can have and memory pools to create `coap_pdu_t` structures. When a new packet is received, the *recvfrom* event returns a pointer to a buffer with the UDP packet received, that can be converted to a `coap_pdu_t` structure. The actual length of each CoAP message depends on the number of CoAP options it contains and of the payload length. Generally, the actual size will be lower than the maximum allowed CoAP message size. Maintaining an unique buffer for sending packets is consuming less RAM than storing a buffer for each CoAP packet. The approach followed is to store received CoAP messages in `coap_pdu_t` structures and use special commands to create a CoAP messages from a `coap_pdu_t` struct and a `coap_pdu_t` structure from a received CoAP message.

In conclusion, to handle UDP message reception the TinyOS *UDP* interface will be used using the `recvfrom` event and `sendto` command. When a new CoAP message is received, the UDP buffer is used to get a `coap_pdu_t` structure and when a new CoAP message is to be sent, the `coap_pdu_t` structure will be transformed into a CoAP message packet. A detailed description of all interfaces is shown in appendix A.

6.2.4 CoAP request/response matching layer

CoAP request/response matching layer is based on two new interfaces: `CoapServer` and `CoapClient`. `CoapServer` interface provides access to commands and events to bind a new UDP server in an specified port, add CoAP server resources and notify that a new response has been created. *CoapClient* interface provides commands and events to send CoAP requests and notify that a new CoAP response has been received.

A CoAP server must have several resources that can be queried to get some information from the device. A new TinyOS interface is defined for this purpose called `CoapResource`. When a new CoAP server component is initialized, the `CoapResource` interface is used as a parametrized interface[27]. For each resource to be used, an unsigned integer is used to identify each `CoapResource` component wired to the parametrized interface. A parametrized interface can be seen as a list of implementations of the `CoapResource` interface each of them identified by an unsigned integer key.

`CoapServer` interface has a command to initialize it which allows to specify the port to be binded and a list of CoAP resources to add to the server. A CoAP resource is defined as follows:

```
typedef struct {
    uint8_t key;
    uint8_t uri[MAX_URI_LEN];
    uint8_t len;
    uint8_t rt[10];
    uint8_t iff[10];
    uint8_t sz;
    uint8_t ct;
    uint8_t is_subscription;
} coap_resource_t;
```

A list of `coap_resource_t` structures is passed to the `CoapServer` interface during initialization. The keys to be used in the parametrized interface mentioned above must be the same that *key* fields used in the structure. When a new CoAP request is received, the server looks for the *Uri-Path* option and tries to match an existing `coap_resource_t` using the *uri* field. If a resource is matched, then the *key* field is used to call the corresponding `CoapResource` implementation using the parametrized interface. For a detailed explanation with examples of how to create a CoAP server application consult **appendix A**.

In figures 6.3, 6.4 and 6.5 there are three flow charts explaining how the CoAP server works. There are two QueueC components: one for the incoming connections (**IncomingQueue**) and other for processing connections (**ProcessQueue**). When a new UDP packet arrives, the server saves it in the **IncomingQueue**. For each new UDP packet received, a new task for processing incoming packets (**processIncoming**) is posted. When a **processIncoming** task is executed, the server takes the head element of the queue and checks which type of CoAP message it has. If a CON or NON message has been received, a new task (**processingResources**) to handle the request is posted. This task executes the corresponding method (GET, POST, PUT or DELETE) on a resource. When the server finishes processing the request, a new event is signaled (**isDone**) providing the response as a parameter.

If the response is a CON message (because it is a separate response), the response is stored in a list (*pending_list*) and a Timer component (**Timer1**) is started. If not, the response is sent directly to CoAP client. RESPONSE_TIMEOUT and MAX_TRANSMIT constants are used in the program. The `fired` event of **Timer1** is signaled when RESPONSE_TIMEOUT seconds have passed. It looks how many times the response has been retransmitted and if it is less than MAX_TRANSMIT, it updates the retransmission time out for the specified response. If MAX_RETRANSMIT retransmissions have been done, then it deletes the response from the *pending_list*. If the server is not able to process a CON message, a RST message is sent.

If an ACK is received, the server tries to search in *pending_list* for a previously sent response that matches the received ACK. If found, it is deleted from the list. If a RST is received, the server searches in *pending_list* for a previously sent response that matches the received RST message and, if found, it is deleted immediately.

This concrete CoAP server implementation does not support proxying.

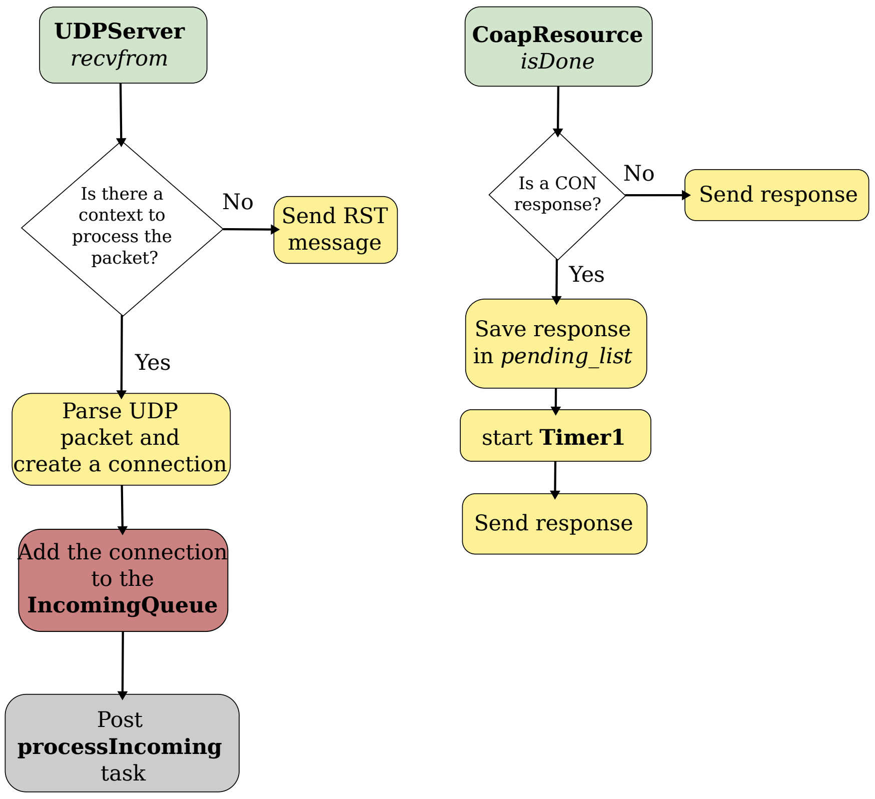

UDPServer
recvfrom
Is there a context to process the packet?
No
Send RST message
Yes
Parse UDP packet and create a connection
Add the connection to the IncomingQueue
Post processIncoming task
CoapResource
isDone
Is a CON response?
No
Send response
Yes
Save response in pending_list
start Timer1
Send response

Figure 6.3: CoAP server flow chart part I

CoapClient interface uses the *sendRequest* command to send a CoAP message to a destination. The destination address is included in the `coap_pdu_t` pointer passed to the `sendRequest` command. In this case there is only a QueueC component for incoming connections (**IncomingQueue**). For each CoAP request sent, a new message ID and token are generated using an internal unsigned integer counter. If the request sent corresponds to a CON message, it is stored in a list (*pending_list*). There

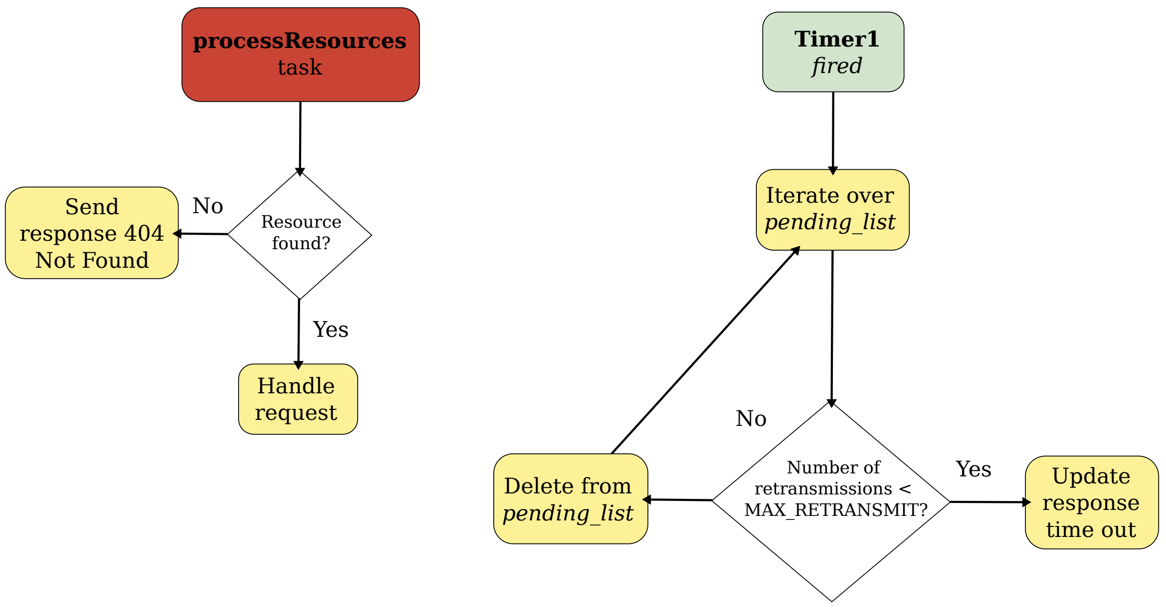


Figure 6.4: CoAP server flow chart part II

is a Timer component that signals the `fired` event when RESPONSE_TIMEOUT seconds have passed. It looks how many times the request has been retransmitted and if it is less than MAX_TRANSMIT, it updates the retransmission time out. If MAX_RETRANSMIT retransmissions have been done, then it deletes the request from the *pending_list*.

When a new response is received it is stored in the (**IncomingQueue**) and the **processIncoming** task is posted. When the **processIncoming** task is executed it looks for the message type of the response received and tries to find a stored requests that has the same token than the received response. If it is a CON or NON response it removes the request from the *pending_list* list to avoid retransmissions. If it is an ACK response also the message ID must match. If the ACK code is different from 0 then means that is piggy-backed response and the `messageReceived` event is signaled. If the response code is 0 it means that is a separate response and the client waits to signal the `messageReceived` event.

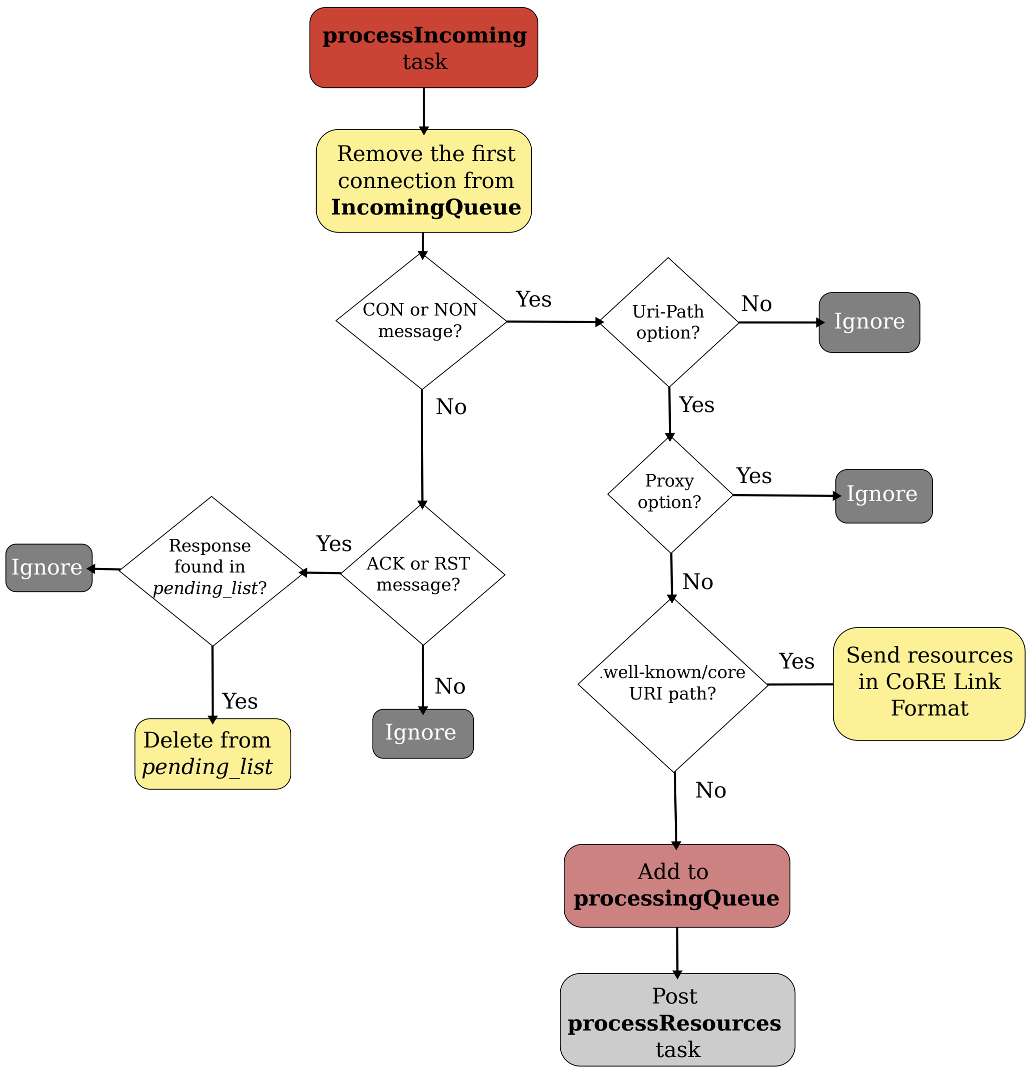


Figure 6.5: CoAP server flow chart part III

7 CoAP tools

In this section a new framework to use CoAP in C applications is presented. Its name is CoAP tools and provides a C library with the following features:

- **CoAP packet handling**. The framework provides the necessary tools to easily handle CoAP packets.
- **CoAP client interface**. It hides the underlying complexity of a CoAP client (retransmissions, request/response matching, etc...). It is also capable of caching CoAP responses.
- **CoAP server interface**. It enables CoAP services in C applications. It also offers features such as service discovery and proxying.
- **CoRE link format support**. Provides the necessary tools to parse CoAP payloads in CoRE link format.

In figure 7.6 a general overview of the framework is shown. It is based on a two-layer model where the first layer deals with CoAP messages and the second provides CoAP services.

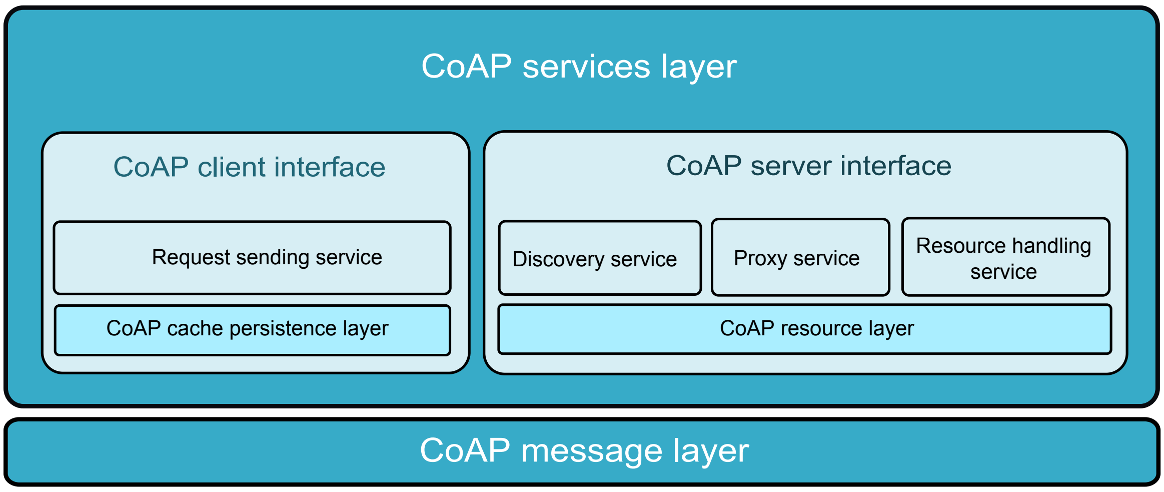


Figure 7.6: CoAP framework overview

CoAP client interface allows an application to send and receive CoAP messages. In order to use the CoAP client interface, the `coap_client_t` structure is used. It provides methods to send requests, to receive responses and to retransmit and clean CON requests pending to be acknowledged.

CoAP server interface allows applications to serve CoAP requests. In order to use the CoAP server interface, the `coap_server_t` structure is provided. It provides methods to add and remove server resources, receive requests, send responses and to clean and retransmit CON responses pending to be acknowledged. It also provides methods to maintain a CoRE Resource Directory[18].

Next subsections provide a more detailed view of the framework, explaining the core parts.

7.1 Generic tools

This section provides a list of generic C structures and functions used by CoAP tools. For a more detailed information about all the available functions, see appendix C.

7.1.1 CoAP list

One element used by CoAP tools applications is a like the one presented in subsection 6.2.3 but with some small differences. Structures and functions used to handle a linked list are defined in `coap_list.h` header file (See appendix C.2). A linked list in CoAP tools is defined by the following C structures:

```
struct coap_list_t {
    struct coap_node_t *first;
    struct coap_list_index_t  iterator;
    void (*delete_func)(struct coap_node_t *);
    int (*order_func)(void *, void *);
};

struct coap_node_t {
    struct coap_node_t *next;
    char *key;
    void *data;
};

struct coap_list_index_t {
    struct coap_list_t *list;
    struct coap_node_t *this, *next;
};
```

A *coap_list_t* structure wraps a linked list as the one shown in figure 7.7. Each element inside the linked list is a `coap_node_t` structure that has a pointer to the next element in the list. Each node has a void pointer (*data*) that points to the real location of the data and a unique key (*key*) to identify it. As the data and the key are both void pointers, they can be of any type making the list generic.

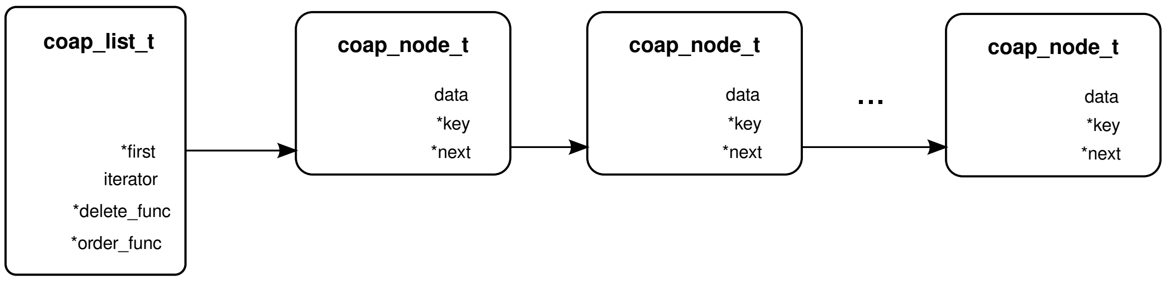


Figure 7.7: Linked list

The following methods are provided in order to work with a `coap_list_t` structure:

- ***coap_list_create***: Creates a new list.
- ***coap_list_node_create***. Creates a new list node.
- ***coap_list_node_insert***. Adds a new node to the list.
- ***coap_list_node_delete***. Removes a node from the list.
- ***coap_list_clean***. Cleans the list. It removes all nodes inside the list but it does not free the list.
- ***coap_list_delete***. Removes the list.

One common operation will be iterating over the list. For this purpose, the iterator structure `coap_list_index_t` shown above is used. A list returns an iterator structure using the `coap_list_first` function. For each iteration state, the function `coap_list_this` returns the current node. Finally, the function `coap_list_next` updates the iteration state or returns NULL if the end of the list is reached. One brief example that shows how to use an iterator is the following:

```
coap_list_index_t *li;
void *data;

for (li = coap_list_first(res->connections);li; li = coap_list_next(li)) {
    coap_list_this(li, NULL,(void **) &data);
    // DO SOMETHING WITH DATA,  EVEN DELETE THE ELEMENT
}
```

For each iteration state, node's data is obtained with the function `coap_list_this`. One important remark is that the element obtained in the current iterating state can be safely removed from the list.

7.2 CoAP message layer

CoAP packet creation and manipulation is the most essential part. It allows to read CoAP packets received from an UDP socket and create C structures that contain the necessary CoAP PDU fields. The following C structures have been defined to wrap a CoAP PDU defined in 3.2.2:

```
typedef struct {
    coap_hdr_t hdr;
    coap_list_t *opt_list;
    char payload[COAP_MAX_PAYLOAD_SIZE];
    unsigned int payload_len;
    struct sockaddr_in6 addr;
    time_t timestamp;
} coap_pdu_t;

typedef struct {
    char version;
    unsigned char type;
    unsigned char code;
    unsigned char optcnt;
    unsigned short id;
} coap_hdr_t;


typedef struct {
    char code;
    int len;
    char data[MAX_OPT_DATA];
} coap_option_t;
```

- **hdr**: abstracts the CoAP PDU header. It contains the CoAP version, message type, message code, the number of options and the message ID.
- **opt_list**: contains a linked list (see section 7.1.1) with CoAP options. Each option is wrapped by a `coap_option_t` structure that contains the code, the length and the value of the option.
- **payload**: contains a buffer to store the PDU payload. Its size is the maximum allowed packet size `COAP_MAX_PAYLOAD_SIZE` defined in `coap_general.h` (See C.3).
- **payload_len**: the payload length.
- **addr**: In case of a request contains the source address of the received UDP packet. In case of a response contains the destination address where to send the UDP packet.

- **timestamp**: contains a time stamp to know when the CoAP PDU was created.

CoAP tools provide several functions to deal with `coap_pdu_t` structures. Header definitions can be found in the `coap_pdu.h` header file (see C.6). There are two functions to retrieve and create a CoAP packet: `coap_pdu_packet_write` and `coap_pdu_packet_read`.

The first function fills a buffer with a CoAP packet using a `coap_pdu_t` structure. This buffer will contain a CoAP packet which is ready to be sent in the payload of an UDP packet. With this strategy, the buffer used to send the CoAP PDU is independent of the real structure that maintains the data, thus it is possible to reuse the same buffer to send different CoAP packets. The length of the created packet is also returned by this function.

The second function creates a new `coap_pdu_t` structure from a received CoAP packet. Once created, its information can be easily read and modified using structure fields and methods provided by the framework. In particular, the following actions can be performed over a CoAP PDU:

- **Get and modify CoAP version**: using the `hdr.version` field.
- **Get and modify CoAP message type**: using the `hdr.type` field.
- **Get and modify CoAP message code**: using the `hdr.code` field.
- **Get and modify CoAP options number**: using the `hdr.optcnt` field.
- **Get CoAP option**: CoAP options can be returned as a `coap_option_t` structure. The `coap_pdu_option_get` method returns a CoAP option an option code. Option codes are defined in `coap_option.h` (See C.5).
- **Add CoAP option**: a new CoAP option can be added to a CoAP PDU with the method `coap_pdu_option_insert`. It automatically increments the `hdr.optcnt` field.
- **Delete CoAP option**: a new CoAP option can be deleted from a CoAP PDU with the method `coap_pdu_option_unset`. It automatically decrements the `hdr.optcnt` field.
- **Delete all CoAP options**: all CoAP options can be deleted from a CoAP PDU with the method `coap_pdu_option_delete`. It automatically sets the `hdr.optcnt` field to zero.
- **Get and modify CoAP payload**: access the `pdu->payload` field. To copy a new payload the `sprintf`[2] standard function can be used.
- **Get and modify CoAP payload length**: access the `pdu->payload_len` field. This field must be set to the correct payload length. If set to zero, no payload will be sent.
- **Get and modify CoAP source or destination address**: access the `pdu->addr` which contains a sockaddr_in6 structure.

Some extra useful tools are also provided:

[2]See http://linux.die.net/man/3/sprintf or use `man sprintf` for further information.

- **Given a CoAP URI get destination address, path and query string**. It is useful(is case of sending the proxy option for instance) to extract the destination information given a CoAP URI. The function `coap_pdu_parse_uri` provides this functionality.

- **Given a query string variable determine if exists in the global query string and return its value**. When a query string is sent (for instance to perform searches in a resource directory) is useful to determine if some parameter exists and its value. The function `coap_pdu_parse_query` provides this functionality.

7.3 CoAP services layer

7.3.1 CoAP context

CoAP client and server structures use what in the framework terminology is called CoAP context. The idea behind is that a CoAP context has the basic components to offer server and client services. A CoAP context C structure is defined as follows:

```
struct coap_context_t {
    int socket;
    coap_list_t *pending_queue;
    coap_list_t *dispatch_queue;
    unsigned int message_id;
    unsigned int token;
};
```

- **socket**: each context has an UDP socket that is used for sending and receiving CoAP packets.

- **dispatch_queue**: All the incoming CoAP UDP packets are placed in the dispatch queue for later processing.

- **pending_queue**: All the CoAP UDP packets pending to be acknowledged are placed in the pending queue.

- **message_id**: If a shared socket is used to send multiple CoAP messages a way to generate unique IDs must be used. In this case a simple approach is followed: `message_id` field is a counter which initially contains a random number but with each CoAP message sent it is incremented. This way the message ID will be unique until it reaches the maximum value an unsigned integer.

- **token**: Multiple clients can use the same context and therefore the same socket. A token is needed in each request to match it with its corresponding response. An unsigned integer counter is maintained for this purpose using the same approach than the CoAP message ID generation.

7.3.2 CoAP client service

CoAP client services provide external C programs the ability to easily send CoAP requests to CoAP servers. It handles the underlying complexity of a CoAP client: request/response matching, token and message ID generation, etc... CoAP client services are based on a client C structure as follows:

```
typedef struct {
    coap_context_t *context;
    void (*response_received)(coap_pdu_t *, void *);
    void (*pending_removed)(char *, void *);
    DB *cache_db;
    void *data;
} coap_client_t ;
```

A CoAP client structure uses a `coap_context_t` structure in order to get a context to send and receive CoAP messages. When a new CoAP response is received and matched with a previously-sent request, the `response_received` callback function is called. Also the `pending_removed` callback function is called when a request is removed from the pending list because the maximum number of retransmissions is reached or because a response that matches the request has been received. When these callbacks are called two parameters are passed: the CoAP response (*response*) and an optional void pointer (*data*).

In order to enable caching capabilities in the CoAP client, a Berkeley[28] database is used to store CoAP requests. A database object `DB` is passed when a `coap_client_t` structure is created.

The framework provides the `coap_client_send` function in order to send a CoAP PDU. To enable packet reception and to clean and resend CoAP PDU requests, there are three functions:

- **coap_client_read**: reads incoming CoAP responses and places them in the dispatch queue.
- **coap_client_dispatch**: dispatches the receives CoAP responses and performs the request matching and calls `response_received` or `pending_removed` whenever necessary.
- **coap_client_clean_pending**: handles CoAP request retransmission and cleans expired requests that reached the maximum number of retransmissions.

In order to use these functions, a UNIX selector can be used to call `coap_client_clean_pending` function for instance each second. The following code is a brief C example of how to use CoAP client C structures:

```
timeout.tv_sec = 1;
timeout.tv_usec = 0;
do {
    rc = select(max + 1, &lset, NULL, NULL, &timeout);
} while(rc < 0 && errno == EINTR);
if(rc == 0) {
    continue;
}
if (FD_ISSET(client->context->socket,&lset)) {
    coap_client_read(client);
    coap_client_dispatch(client);
} else {
    coap_client_clean_pending(client);
}
```

The CoAP context socket must be added the list of sockets that selector is watching[3] (in this particular example *lset*). The *timeout* variable specifies a time out to unblock the selector calling when nothing is sent through the socket.

When the `select` calling is unblocked is because a new UDP packet is received or because the time out has expired (in this particular example 1 second). In the first case, the client needs to handle the CoAP packet reception and dispatching. This is accomplished using the `coap_client_read` and `coap_client_dispatch` functions. In the later case, the client needs to see if some CON request needs to be resent or removed from the pending list. This is accomplished using the function `coap_client_clean_pending`.

Figure 7.8 gives a high-level overview of the algorithm used when receiving a CoAP packet.

[3]See http://linux.about.com/od/commands/l/blcmdl2_select.htm or use *man 2 select* to get further information about UNIX select system call.

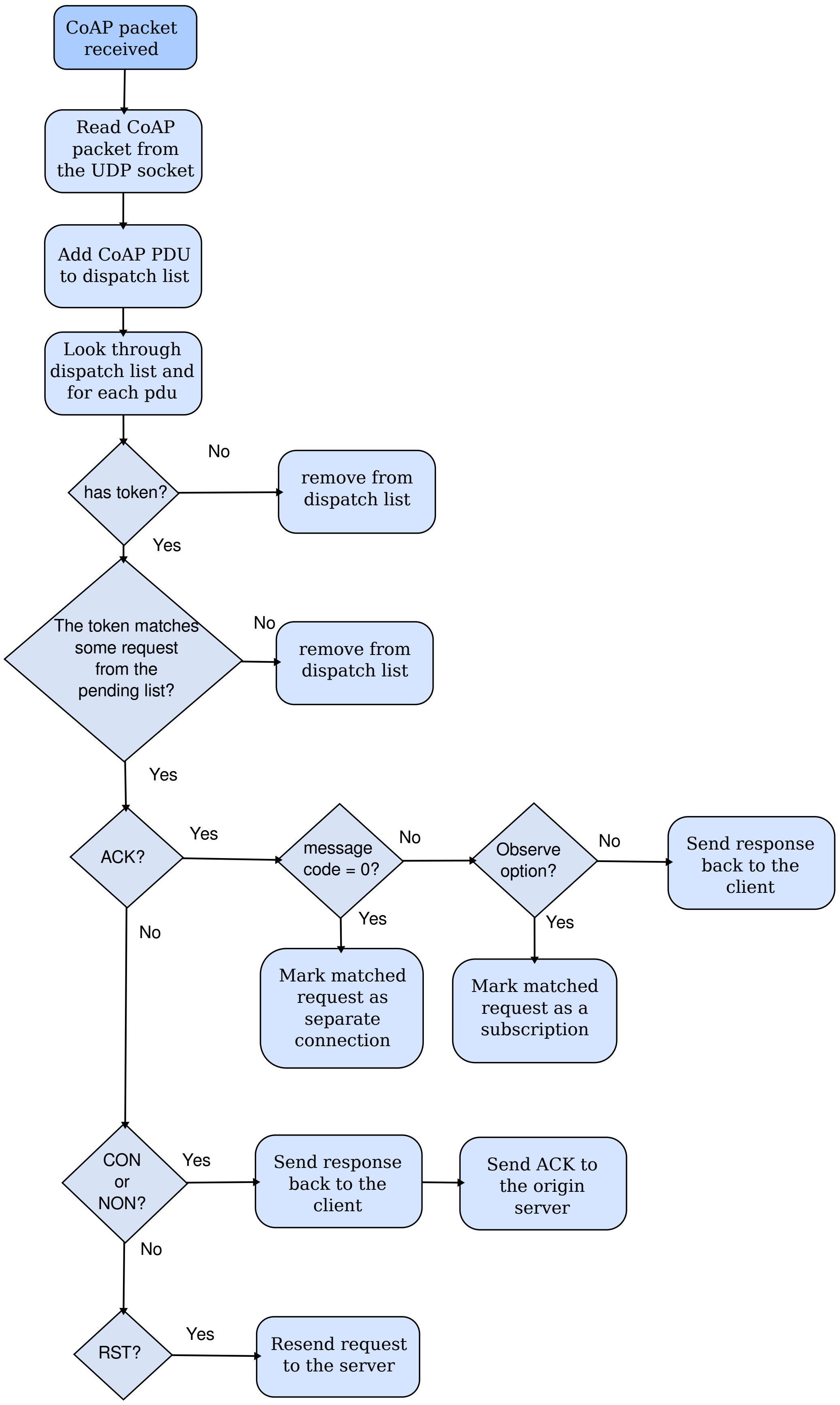

CoAP packet received
Read CoAP packet from the UDP socket
Add CoAP PDU to dispatch list
Look through dispatch list and for each pdu
has token?
No
remove from dispatch list
Yes
The token matches some request from the pending list?
No
remove from dispatch list
Yes
ACK?
Yes
message code = 0?
No
Observe option?
No
Send response back to the client
Yes
Mark matched request as separate connection
Yes
Mark matched request as a subscription
No
CON or NON?
Yes
Send response back to the client
Send ACK to the origin server
No
RST?
Yes
Resend request to the server

Figure 7.8: CoAP client overview

7.3.3 CoAP server service

CoAP server services provide external C programs the ability to easily integrate a CoAP server. It handles CoAP request reception and response sending and lets programmers to define resources to handle incoming requests. It also enables external servers to register in a resource directory and clients to perform resource-based querys[18]. It can also connect to CoAP servers on behalf of a client providing proxying capabilities.

CoAP server service implements the following:

- Constrained Application Protocol[6].
- CoRE Link Format[29].
- Observing Resources in CoAP[30].
- CoRE Resource Directory[18].

The CoAP server object is as follows:

```
typedef struct {
    coap_context_t *context;
    coap_list_t *resources;
    coap_list_t *subscriptions;
    coap_list_t *proxy;
    coap_rd_t *rd;
    int (*rd_update)(int, coap_rd_entry_t *, coap_resource_t *);
} coap_server_t ;
```

- **context**: a `coap_context_t` structure. In this case the dispatch queue will contain incoming requests and the pending queue responses pending to be acknowledged.
- **resources**: the list of resources available in the server which are not subscriptions.
- **subscriptions**: the list of resource subscriptions available in the server. Each resource subscription is a C struct that basically wraps a CoAP resource and a list of observers.
- **proxy**: the list of requests that request a proxy connection. Proxy requests need to be stored separately to forward the received response to the correct client host.
- **rd**: a CoAP resource directory structure. It stores `coap_rd_entry_t` C structures, one for each node available which is registered in the directory.

In the framework terminology a CoAP resource is a service that the server can provide through an URL generating a response. CoAP resource C structures are defined as follows:

```
struct coap_resource_t {
    char *path;
    char *name;
    char *rt;
    char *ifd;
    int sz;
    struct sockaddr_in6 server_addr;
    unsigned int is_subscription;
    void (*handler)(coap_pdu_t *);
};
```

A CoAP resource contains a description based on CoRE Link format[29]: resource type, interface description and estimate maximum size. Each resource is identified by a path that is the URL to access the service. A name is also used to give a description to the resource. A resource also contains the server address (a `sockaddr_in6` structure) and specifies if the resource handles subscriptions or not using the `is_subscription` field .

Each resource uses a handler callback to generate a response given a request. Thus, each time a request is sent, the requested resource calls the *handler* callback function which returns the response with a `coap_pdu_t` structure pointer given as a parameter. It is important to remark that the `coap_pdu_t` pointer structure given as a parameter is a pointer to the request PDU, and the response must be created modifying the structure pointed by the pointer. This makes the response generation easier because most of the responses will echo the TOKEN option and the message ID. Thus, to create a response basically the message type and code should be changed, some options deleted and a new payload copied.

A CoAP server maintains a list of CoAP resources. When a new request is received the server tries to match a resource using the URI PATH option. If matched, it calls the `handler` function and sends the response back.

When a request which tries to establish an observation relationship is received, the server tries to match a resource from the subscriptions list. To model a subscription, a C structure has been defined:

```
typedef struct {
    coap_resource_t *resource;
    coap_list_t *observers;
    unsigned int observe;
    time_t last_timestamp;
} coap_server_subscription_t;
```

- **resource**: the resource that supports subscriptions.
- **observers**: a list with all clients that must be notified whenever the resource state changes.
- **observe**: an unsigned integer is maintained to update the OBSERVE option.

- **last_timestamp**: when the last notification was received. Used for message reordering.

Another important capability provided is the CoRE resource discovery[18]. Each server can be used to allow clients to find which resources are available in the 6LoWPAN network. Figure 7.9 shows an example of a CoAP client requesting temperature resources in the 6LoWPAN consulting a resource directory.

In order to maintain the directory, CoAP end-points can send their available resources in CoRE link format sending a POST request to the CoAP server that contains the resource directory. They can also update the previously-sent resource list using a PUT request and delete them using a DELETE request.The CoAP server that has the resource directory responds with a CoAP response in CoRE link format.

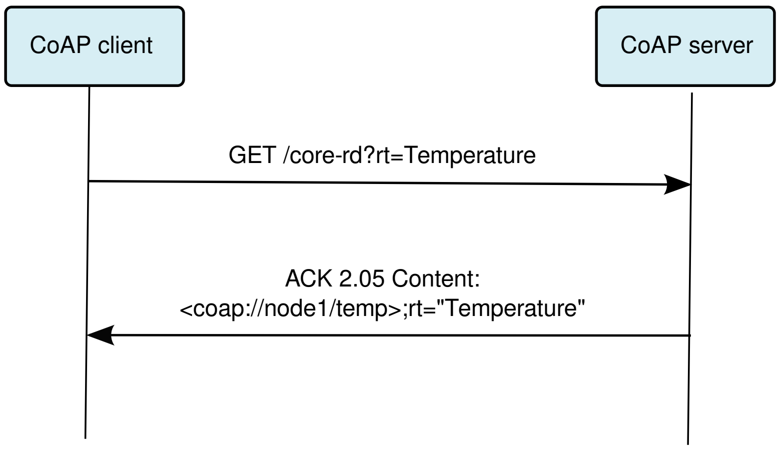

CoAP client
CoAP server
GET /core-rd?rt=Temperature
ACK 2.05 Content:
<coap://node1/temp>;rt="Temperature"

Figure 7.9: CoAP client requesting temperature resources to a CoAP server with a CoRE resource discovery.

To manage directory entries the framework uses the following C structures:

```
typedef struct {
    char *name;
    int lt;
    int id;
    coap_list_t *resources;
} coap_rd_entry_t;

typedef struct {
    coap_list_t *entries;
} coap_rd_t;
```

A CoRE resource discovery directory structure (`coap_rd_t`) contains a list of CoAP entries which are identified by a *name*, *id*, *lt* and is intended to identify a CoAP node. Each entry contains a list of resources. When a new resource is added to an entry the callback function `rd_update` function is called providing pointers to the modified entry and the resource added or modified.

To support CoAP resource discovery and also to provide a list of the available resources a server has, the CoRE link format is used. The header file `coap_resource.h` provides two important functions to handle this format:

- ***coap_resource_link_format***: gets the CoRE Link representation of a resource.
- ***coap_resource_update***: updates a list of resources with the resources provided in CoRE link format. If some resource was existing, it is ignored.

Figure 7.10 show a high-level overview of the algorithm used by to provide the CoAP server service.

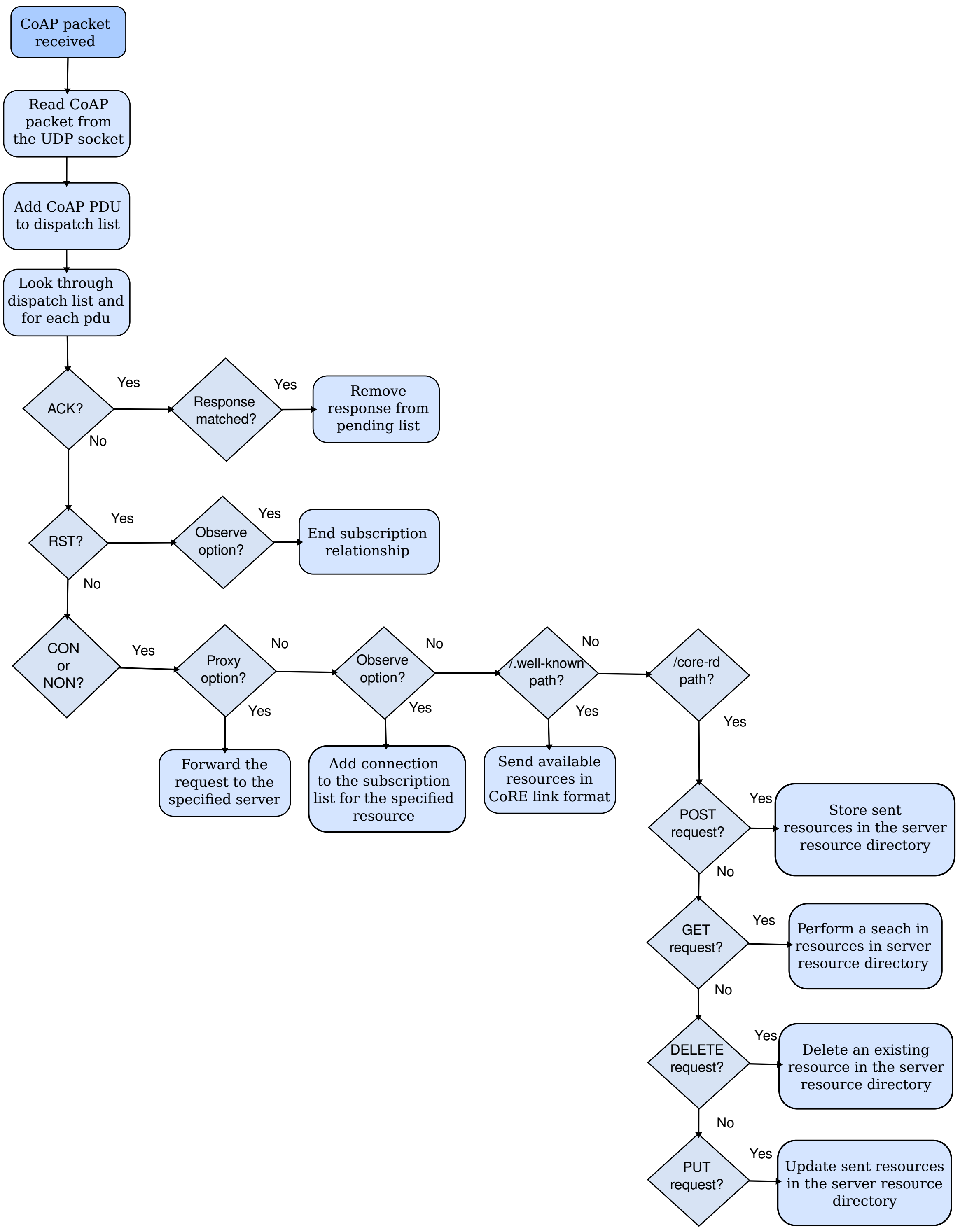

CoAP packet received
Read CoAP packet from the UDP socket
Add CoAP PDU to dispatch list
Look through dispatch list and for each pdu
ACK?
Yes
No
Response matched?
Yes
Remove response from pending list
RST?
Yes
No
Observe option?
Yes
End subscription relationship
CON or NON?
Yes
Proxy option?
No
Yes
Forward the request to the specified server
Observe option?
No
Yes
Add connection to the subscription list for the specified resource
/.well-known path?
No
Yes
Send available resources in CoRE link format
/core-rd path?
Yes
POST request?
Yes
Store sent resources in the server resource directory
No
GET request?
Yes
Perform a seach in resources in server resource directory
No
DELETE request?
Yes
Delete an existing resource in the server resource directory
No
PUT request?
Yes
Update sent resources in the server resource directory

Figure 7.10: CoAP server service overview

8 CoAP Web services implementation on GW2358-4 Cambria network computer

This section presents the requirements, hardware and software selected to provide CoAP Web services on GW2358-4 Cambria network computer[31].

8.1 Proxy requirements

First, the software requirements must be defined. In previous sections a general vision of the software requirements was provided. In this section a detailed requirement list is provided:

- **HTTP message handling**. The proxy must be able to receive and process HTTP connections. This is accomplished extending a Web server functionality. Possible options to do this were analyzed and discussed in 5.3.
- **CoAP message handling**. The software must be able to handle CoAP messages. CoAP tools framework will be used for this purpose. This includes also caching capabilities and services such as proxying, resource discovery or subscriptions.
- **Administration**. The software should allow a user to modify its behavior. A solution is proposed in 8.3.4.

8.2 Hardware

Hardware is the first important decision to make. Depending on the hardware different tools are available. In this work two main options were considered:

- **PC**: Many well-known tools and operating systems available. As it is a good alternative, it normally has more resources than needed.
- **Gateway**: More suitable alternative in this scenario. Processor and memory requirements are lower and the hardware size is smaller. As the processor normally has a different architecture than a PC, cross-compiling tools and a suitable operating system is a requirement.

A PC was only considered at the beginning for testing purposes. The real CoAP proxy is an embedded platform. The hardware used in this work is the *GW2358-4* Cambria Network Computer[31]. The main technical specifications are:

- Intel XScale IXP435 667MHz Processor.
- 128Mbytes DDRII SDRAM Memory.
- 32Mbytes Flash Memory.

- Compact Flash Socket.
- RS232 Serial port, two v2.0 Host USB ports and two 10/100 Ethernet Ports.

This particular hardware boots from a *compact flash* memory card of 4Gb of memory. It is used to store the operating system and the file system. With the Ethernet ports, the gateway can be connected to IP over IEEE 802.3 networks. However, some hardware interface for IEEE 802.15.4 networks is needed. To do this task a simple solution is followed. As the embedded platform has USB ports, a TelosB mote can be attached to them. This mote can be the IEEE 802.15.4 interface and it can communicate with the gateway through the serial interface.

8.3 Software

Once decided the hardware used, some kind of software must be used or created in order to get the required functionality. The goal of this software is to allow Web clients to communicate with CoAP end-points and vice versa. This means, it must provide a way to accept incoming HTTP and CoAP requests. It must be also capable of generating HTTP and CoAP responses.

In figure 8.11 main software components are found. There are five main software components that the CoAP proxy must have:

- **Operating system**. Linux operating system is what is best suited for embedded systems. A lot of GNU toolchains are available for multiple platforms. The Linux operating system used in this work is OpenWrt.
- **6LoWPAN driver**. A driver to communicate with the TelosB attached to the USB port. In recent distributions of TinyOS, the BLIP library provides a 6LoWPAN stack for the motes and also a driver for motes attached to the USB port. This driver is written in C language and can be cross-compiled.
- **CoAP proxy daemon**. As there is no clear solution for WSN Web Services interoperability, a custom software is provided.
- **PHP FastCGI daemon**. A PHP daemon is installed to be able to use PHP applications inside the gateway.
- **PHP admininistration Web application**. Allows to configure the gateway through a Web-based interface.

8.3.1 OpenWrt

The operating system running on top of the *GW2358-4* Cambria Network Computer is OpenWrt[32]. It provides a fully writable file system with package management.

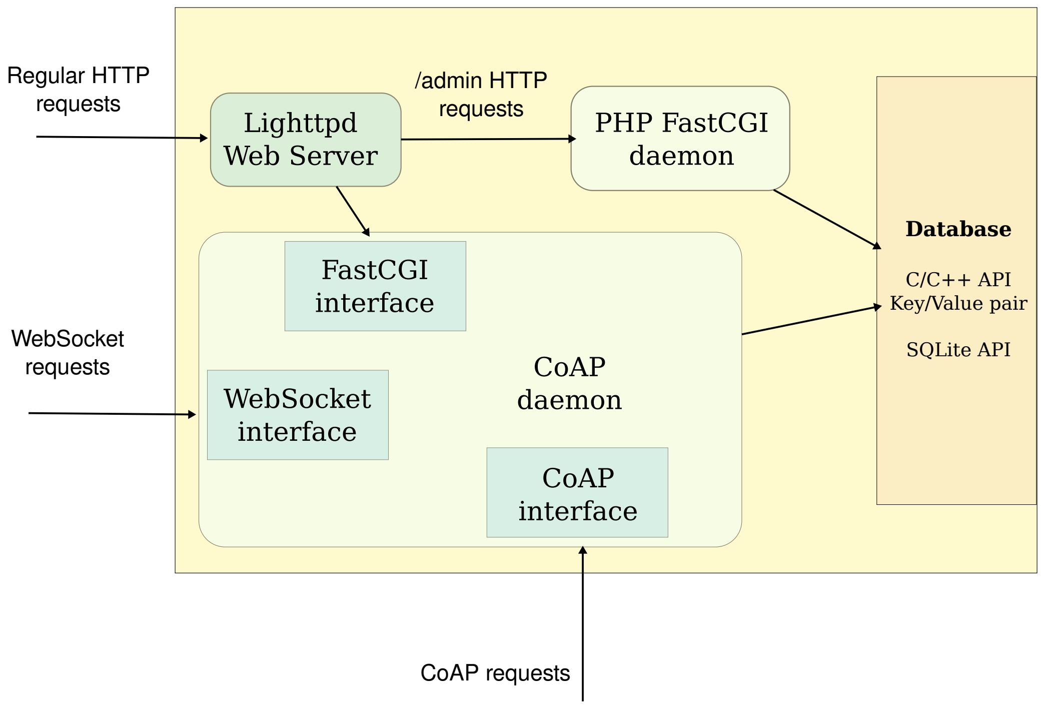


Figure 8.11: General software overview

To write custom applications for the gateway, a cross compiler is needed. OpenWrt provides a development toolchain platform for multiple architectures. In this work the steps described in [33] were used to build the cross-compiling environment. The resulting tool chain built has the name target-armeb_v5te_uClibc-0.9.30.3.

8.3.2 BLIP driver

One important aspect to consider is how to connect to the WSN network. The gateway does not provide an IEEE 802.15.04 interface. In this case a bridge between two networks is necessary.

In this work, the BLIP driver is used. It is part of the BLIP library in TinyOS and provides the basic edge router functionality. In the figure 8.12 an overview of the driver functionality is shown. This solution does not require any extra hardware as TelosB provides a serial interface that can be connected to the USB port. BLIP driver uses TelosB serial interface to receive and send 6LoWPAN packets.

BLIP driver must be compiled using the *target-armeb_v5te_uClibc-0.9.30.3* cross compiler[4].

8.3.3 Lighttpd Web server

Lighttpd is a fast and high performance Web server with a small memory footprint compared with other Web servers. The main aim of the Web server in this work is to handle incoming HTTP requests

[4]The BLIP driver has a small issue in the GW2358-4 network computer. It is not detecting new devices connecting to the network. See appendix B.1 to see the specific compilation and modification instructions.

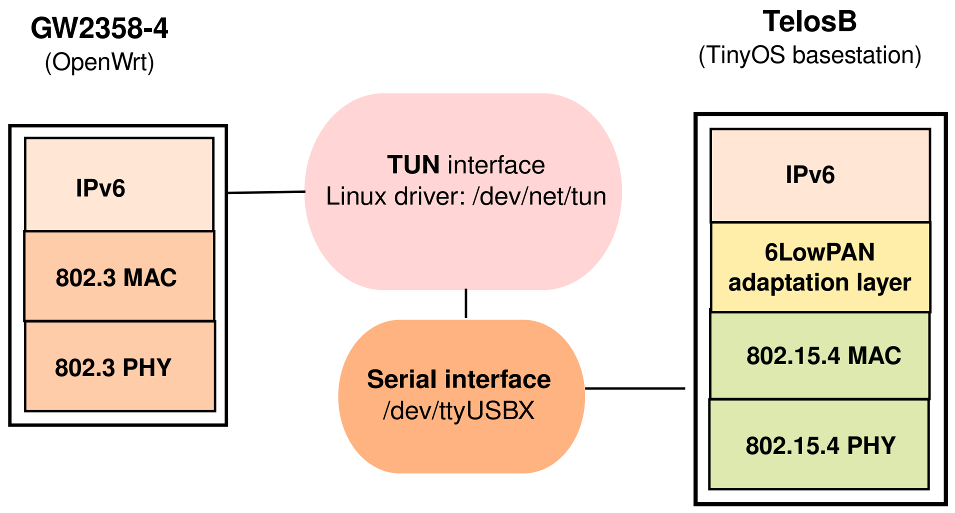


Figure 8.12: BLIP driver schema

and send those requests to a FastCGI daemon.

To send a request to a FastCGI daemon, an URL must be specified. In Lighttpd, an URL for this purpose is defined specifying the UNIX socket that the FastCGI daemon is using. The problem is that to make the proxy transparent to external Web clients, some kind of URL mapping must be used. Lighttpd provides the mod_rewrite module to perform this task. It converts an URL pattern to another URL. It also allows taking some parts of the requested URL and use them for example as a query string for the new URL.

Lets say that the URL to access to the FastCGI daemon is: *http://server-ip/coap.cgi*. In order to make the URL *http://server-ip/node1/example* available, one possibility is to get the URI part of the URL (*/node1/example*) and pass it to the FastCGI daemon using a query string. In this case the Web server would rewrite the URL *http://server-ip/node1/example* to *http://server-ip/coap.cgi?uri=/node1/example*. With this method, the FastCGI daemon can process generic resources requested by Web clients.

A detailed explanation and an example of a Lighttpd Web server configuration is given in the appendix B.2.

8.3.4 Administration

The administration provides a Web interface to control the gateway, visualize data and make connectivity tests. The administration part requires:

- **URI server mapping**. URIs in the gateway must be mapped to URIs in CoAP servers.
- **Connectivity test**. Long polling and Websocket test.
- **Sensor data reading**. Displays the sensor readings from regular requests and subscriptions.

8.3.5 CoAP proxy daemon

In figure 8.14 CoAP proxy software components can be found. It provides several interfaces that are ways to communicate with the daemon. There are three different software interfaces:

- **FastCGI interface**. Regular HTTP requests are received by the CoAP proxy daemon through an *UNIX socket* using the FastCGI protocol. FastCGI protocol is provided by FastCGI libraries[5]. They provide a simple interface to deal with new incoming requests and generate HTTP responses.

  The FastCGI daemon listens on an UNIX socket. When a new request is made to the Web server to a specific URL, it connects to the UNIX socket and sends the request using the FastCGI protocol to the FastCGI daemon. Then, it takes the requests, does some tasks and returns a response using the FastCGI procotol sending it back through the UNIX socket connection.

- **CoAP interface**. CoAP clients can request the CoAP proxy daemon. From their point of view, the daemon is simply a CoAP server. In this case, it provides the following resources:

  1. **Service discovery**: CoAP clients can request the daemon to see which resources are available in the WSN network. They can simply send a GET request to */.well-known/core*. When a new node is inserted in the WSN, it should send a POST request to */core-rd* with its list of available resources in Link CoRE format.
  2. **CoAP-HTTP proxy**: CoAP clients can send a GET request with a *Proxy-Uri-Path* option indicating a HTTP server.

- **WebSocket interface**. Compatible Web browser can send WebSocket requests to the daemon through a prepared TCP port. The service running on this port takes the request, establishes a channel handling the handshake and then the Web browser can receive data through the TCP connection at any time.

[5]For more information visit: http://www.fastcgi.com/drupal/node/6

This software provides two main services:

- **HTTP-CoAP proxying service**. The HTTP-CoAP services allows HTTP clients to send requests to CoAP resources. There are two ways HTTP clients can access to this service: using the FastCGI interface or the WebSocket interface. The first one, is used by the incoming HTTP requests coming from a regular Web server. The Web server communicates with the CoAP proxy daemon using the FastCGI interface. The second one allows compatible Web clients to communicate with the WebSocket interface in order to get access to CoAP node's subscriptions.

  The CoAP proxy daemon listens the possible interfaces using a UNIX *selector*. The CoAP proxy daemon's *selector* listens on the UNIX socket corresponding to the FastCGI interface and on the TCP socket corresponding to the WebSocket interface. The UNIX system call *select* blocks until one of the available channels is ready to read or when a time out expires. When a new connection arrives, *select* unblocks and, depending on the channel ready (UNIX or TCP socket), a different static handler function is called. In this function the CoAP proxy tries to find a resource mapping in the database which URL is the URL by which the client accessed the Web server.

  CoAP proxy has a list with a mapping that relates an URL to access the Web server with a CoAP resource in the resource directory. The proxy maintains a list with all resources that are available through mapping using FastCGI and WebSocket interfaces. This resources, however, are different from CoAP resources presented in CoAP tools. They are defined as follows:

  ```
  typedef struct {
      char *path;
      coap_resource_t *resource;
      coap_list_t *connections;
      coap_client_t *client;
  } resource_t;
  ```

  - **path**: is the path accessible from the Web browser.
  - **resource**: the *coap_resource_t* that this resource is mapping.
  - **connections**: FastCGI or WebSocket connections connected to the resources waiting for a response.
  - **client**: the CoAP client to use to connect with the CoAP server that provides the resource. The client socket defined inside the CoAP context is added to the UNIX selector.

  The available mappings are defined in a SQLite database. Each time a new FastCGI or Web-Socket connection is received, static function named *rebuild_proxy_resources* is called. It updates the list with *resource_t* structures if some change was made in the SQLite database. The SQLite database uses the following tables (see appendix E to see table declaration):

  - **resource**: contains all CoAP resources.
  - **resource_directory**: contains all resource entries in the directory.

– **resource_mapping**: contains mappings between Web server URLs and CoAP resources.

If a mapping is found a new *connection_t* structure is created and added to the *connections* field of the *resource_t* matched. A *connection_t* structure is intended to wrap a generic connection that can be a FastCGI connection or a WebSocket connection. It is defined as follows:

```
typedef struct {
    int type;
    char *path;
    coap_pdu_t *coap_request;
    time_t timestamp;
    void *data;
} connection_t;
```

– **type**: indicates the connection type (FastCGI or WebSocket).

– **path**: is the URL from which the Web client accessed the Web server.

– **coap_request**: the CoAP request generated to contact a CoAP server.

– **timestamp**: a timestamp to know when the connection was created.

– **data**: pointer to optional data used for a specific type of connection. In case of a FastCGI connection the data pointer points to a *FCGX_Request* pointer which is used by FastCGI libraries. In case of a WebSocket connection it contains a pointer to an integer value which is the socket number created when the TCP connection was established.

In figure 8.13 an overview of the HTTP handling can be found.

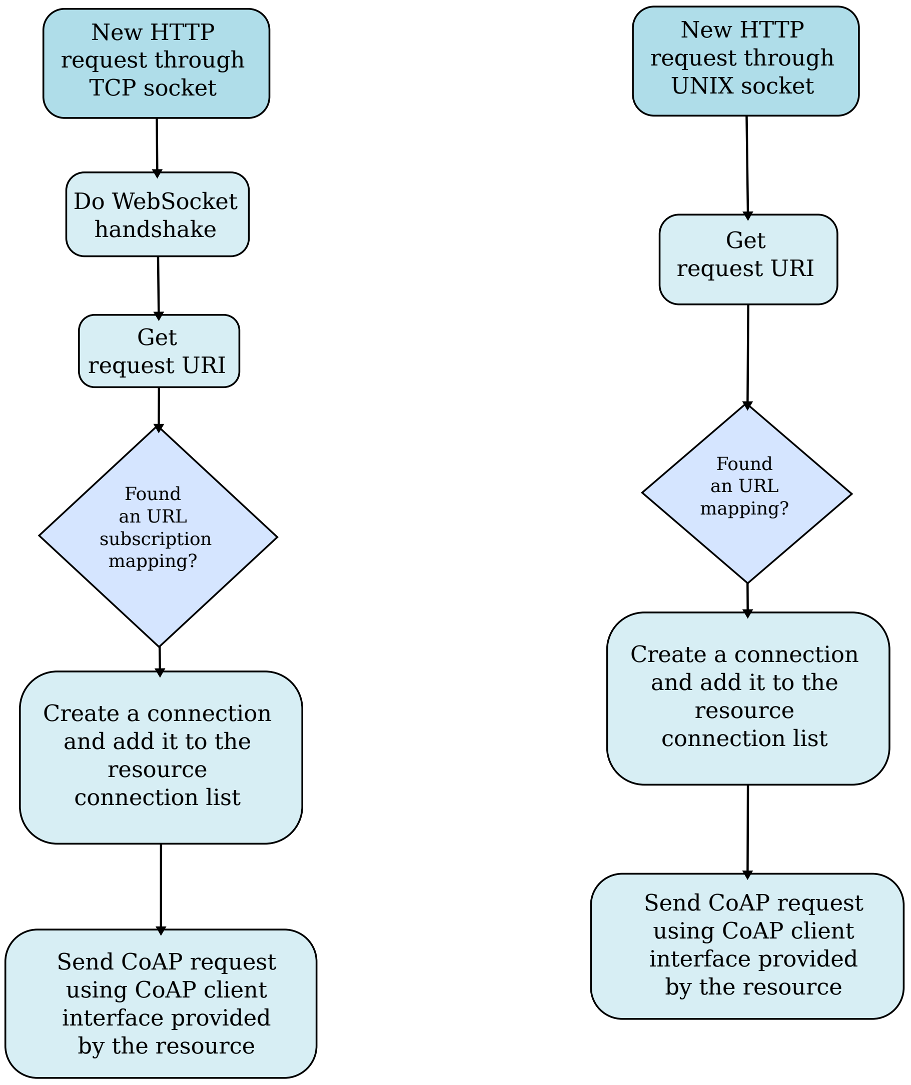

New HTTP request through TCP socket
Do WebSocket handshake
Get request URI
Found an URL subscription mapping?
Create a connection and add it to the resource connection list
Send CoAP request using CoAP client interface provided by the resource
New HTTP request through UNIX socket
Get request URI
Found an URL mapping?
Create a connection and add it to the resource connection list
Send CoAP request using CoAP client interface provided by the resource

Figure 8.13: HTTP request handling

When a new response is received from a socket that some `resource_t` structure has, it checks which connection from its list sent the original request matching the token option. If found, the function `connection_response_send` is called to transform the CoAP response in a FastCGI response or a WebSocket response and return it to the Web browser.

CoAP requests are performed using the CoAP client service from CoAP tools. The optional data passed to this client object is a list of available connections. When a CoAP response is received, the response handler assigned to the client structure uses the extra *data* parameter (which is in this case the list of available connections made) and tries to match the response token with a connection. If matched, the response is sent to the Web browser using the function `connection_response_send`. It handles both the FastCGI and WebSocket cases handling the way the response should be sent in both cases.

- **CoAP services**. From CoAP end-points point of view, the CoAP proxy daemon is CoAP server that provides the services: service discovery and CoAP-HTTP proxying. In fact, this services are provided by the CoAP server service from CoAP tools.

CoAP end-points can connect to external HTTP servers using the CoAP proxy daemon. They must send the *Proxy_Uri* option where its value is the server URL it wishes to connect to. In this case, the CoAP daemon establishes a TCP connection with the destination Web server and waits for a response. When the response is sent back by the destination Web server, it is translated into a CoAP message and sent back to the origin CoAP end-point.

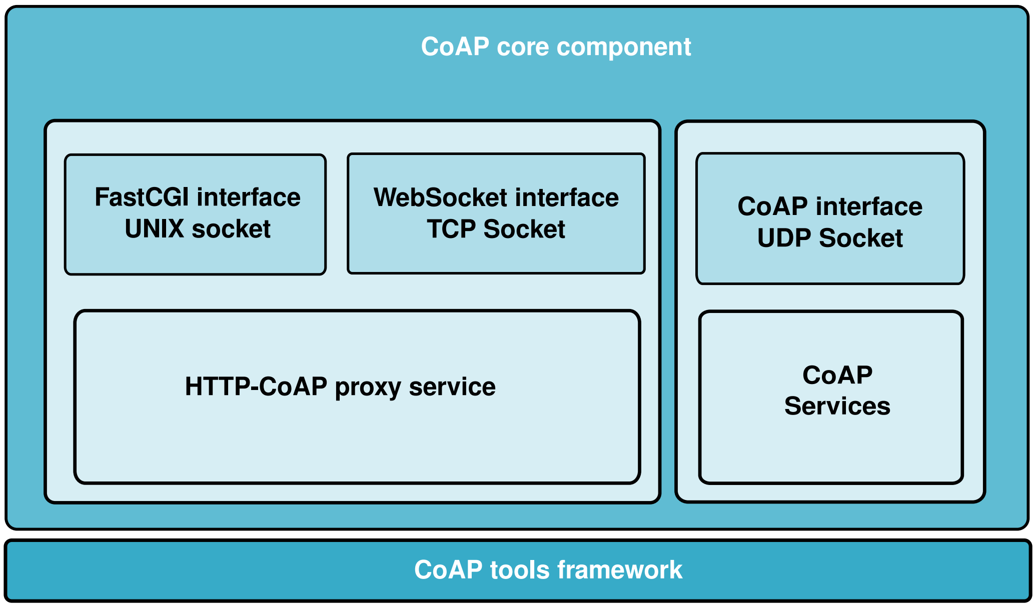

CoAP core component
FastCGI interface UNIX socket
WebSocket interface TCP Socket
CoAP interface UDP Socket
HTTP-CoAP proxy service
CoAP Services
CoAP tools framework

Figure 8.14: CoAP proxy software components

9 Performance evaluation

In this section performance evaluation results are presented. The objective is to evaluate the performance of the developed software.

9.1 CoAP protocol evaluation

This subsection presents an evaluation of the CoAP server performance in terms of latency. Furthermore, it is compared with the performance of a HTTP/1.0 server that uses TCP in the transport layer and with a HTTP server over UDP. Different tests have been performed using several payload values ranging from 10 to 1200 bytes. For each payload value, 100 latency measures have been done. Three test-bed scenarios have been used:

1. **HTTP/1.0 client sending HTTP GET requests to a HTTP/1.0 server. Both are embedded in a TelosB mote**. This scenario evaluates how a HTTP/1.0 server implementation is performing in a single-hop 6LoWPAN network using two TelosB motes. Figure 9.15 shows the described scenario.

2. **HTTP over UDP client sends HTTP GET requests to a HTTP over UDP server. Both are embedded in a TelosB mote**. This scenario evaluates how the HTTP server based on UDP is performing in a single-hop 6LoWPAN network using two TelosB motes. Note that in this case no reliability in the transmission is provided: the HTTP GET request is sent directly in the UDP payload and no reliability mechanism are used in the application layer. In figure 9.15 the described scenario is shown.

3. **CoAP client sends GET CON requests to a CoAP server. Both are embedded in a TelosB mote**. This scenario evaluates how the CoAP server is performing in a single one-hop 6LoWPAN network. Note that in this case, although UDP is used in the transport layer, a simple reliability mechanism is included in application layer. In figure 9.17 the described scenario is shown.

   In this case, the CoAP packet sent in the UDP payload has a total length of 14 bytes. These 14 bytes are divided into the following fields:

   ```
   CoAP HEADER: 5 bytes
   COAP URI_PATH OPTION: 6 bytes
   COAP TOKEN OPTION: 3 bytes
   PAYLOAD: 0 bytes (empty)
   TOTAL: 14 bytes
   ```

   This packet has been selected to compare the same functionality that provides a simple HTTP request. In case of HTTP, the request sent is:

   ```
   GET /test HTTP/1.0\r\n
   ```

```
Host: fec0::2\r\n\r\n
TOTAL: 37 bytes
```

The message sent in the TCP payload has a total length of 37 bytes. The reliability offered by HTTP is provided by the TCP protocol. In case of CoAP, reliability is accomplished with a simple retransmission mechanism implemented in the application layer. The reasons to use these packets to make the evaluation are the following:

- The basic functionality of a HTTP GET request is to get a resource representation which is identified by an URI. This URI is included in the HTTP request line and in this example is */test*. In case of CoAP, to specify an URI the option *Uri_Path* must be used.
- In some cases, CoAP messages share the same communication socket to send multiple messages at once. In order to match a request with a response, some sequence number must be used. In CoAP, both the message ID and the *Token* option are used to match an ACK response. In this particular example, a *Token* option of 2 bytes is included in the CoAP request but it can be increased up to 8 bytes if necessary.

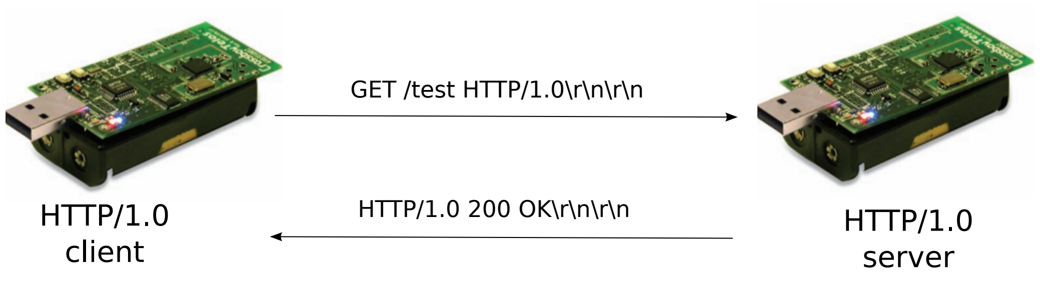


Figure 9.15: **Scenario 1**: HTTP/1.0 client sends GET HTTP requests to a HTTP/1.0 server. Both embedded in a TelosB mote.

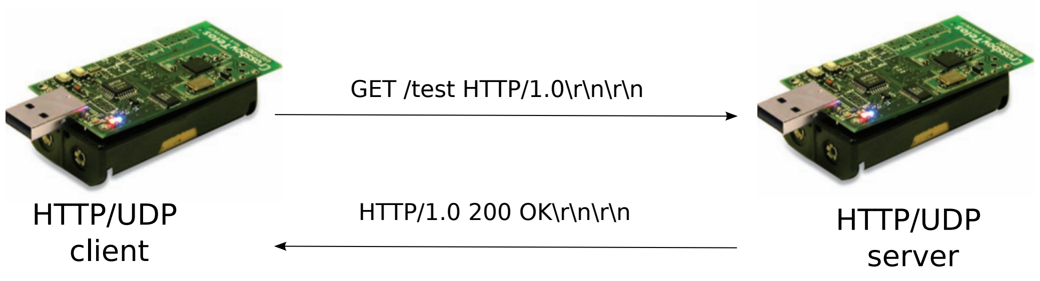


Figure 9.16: **Scenario 2**: HTTP over UDP client sends HTTP GET requests to a HTTP over UDP server. Both embedded in a TelosB mote.

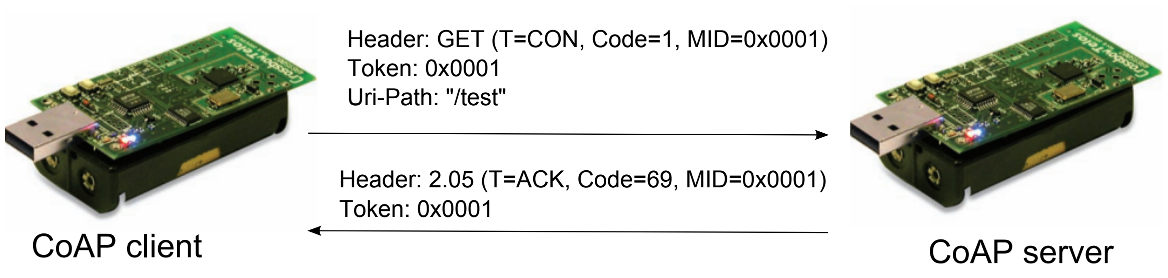

Header: GET (T=CON, Code=1, MID=0x0001)
Token: 0x0001
Uri-Path: "/test"
Header: 2.05 (T=ACK, Code=69, MID=0x0001)
Token: 0x0001
CoAP client
CoAP server

Figure 9.17: **Scenario 3**: CoAP client sends GET CON requests to a CoAP server that responds with an ACK message. Both embedded in a TelosB mote.

Results are presented in figure 9.18.

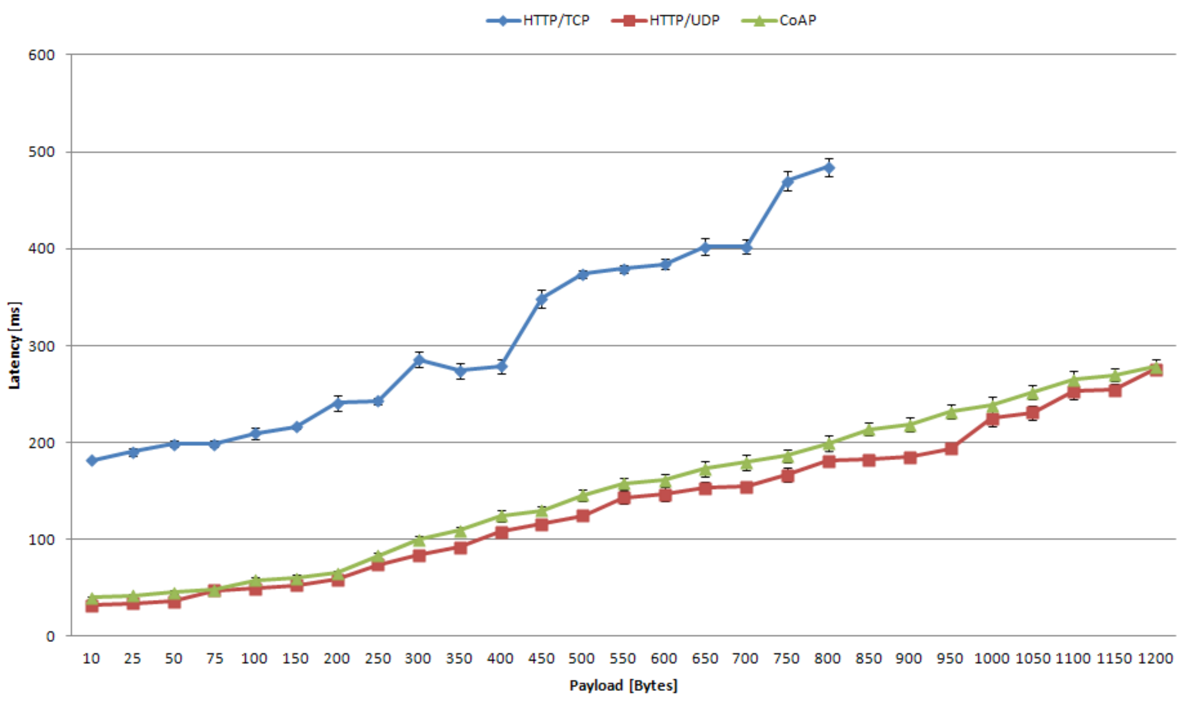


Figure 9.18: Performance evaluation comparing a CoAP server with a HTTP/1.0 server and with a HTTP/UDP server, all embedded in a TelosB mote in a single-hop 6LoWPAN network.

Results show that using a HTTP/1.0 server in a TelosB mote produces higher latency that the other two alternatives. In this case, reliability is accomplished using the TCP protocol in the transport layer. In this particular scenario, it supposes a high overhead in the transmission producing a high packet latency. In this case the latency introduced by the three-way handshake is included in results. The high number of messages involved in a TCP communication make it more prone to suffer retransmissions and consequently to have a significant variation of the delay performance.

The second scenario uses a HTTP/UDP server that does not provide reliability in the application layer. Thus, as the transport layer uses UDP this transmission is not reliable. In fact, using this HTTP/UDP server gets the lowest latency that can be obtained: the server is only sending a response when a requests comes and doing nothing else. On one hand, using HTTP/UDP in this scenario produces low latency due to its low complexity. On the other hand, it is not suitable for other scenarios where a reliable transmission or other features such as request/response matching or packet reordering should be used.

As CoAP is based on UDP, better results than in the HTTP/1.0 case are expected. In fact, results show that CoAP has lower latency. The worst case to consider is when the HTTP/1.0 server performs better (has the lowest latency). The biggest difference between HTTP/1.0 case and the HTTP/UDP case can be seen as an *interval of improvement*. If latency results (even in scenarios where there are packet retransmissions) are inside this interval, it means that the communication improves in comparison with using a HTTP/1.0 server.

Latency results obtained using the CoAP protocol are below the HTTP/1.0 server latency results, which means that CoAP performs better in this environment. Moreover, results obtained are close to the HTTP/UDP case which means that the CoAP server adds low latency during the message processing. Using CoAP has benefits over HTTP/UDP in other scenarios where features such as packet reordering, request/response matching or basic reliability are requirements. As a result, CoAP is a good alternative for WSNs.

9.2 CoAP proxy evaluation

This section presents the evaluation of the CoAP proxy which provides CoAP and CoAP/HTTP mapping services. The objective of this subsection is to evaluate the impact of the using this proxy in an edge router in a single-hop 6LoWPAN network.

9.2.1 CoAP services

This subsection presents and discusses the main improvements offered by CoAP services:

- **CoAP proxy service**. Allows CoAP end-points to contact CoAP servers using the the edge router as an intermediary. When a node is closer to the edge router than to the server it wishes to contact, then CoAP proxying service combined with caching can be employed to reuse some stored CoAP responses and to avoid contacting the requested server. This improves the overall latency.

- **Resource directory**. Allows CoAP end-points to know which devices are present in the network and their capabilities. In this way, CoAP end-points that need some information can request the edge router for some nodes that can help them. The edge router can request other CoAP end-points to know their capabilities using their *./well-known/core* resource. However, according to [**18**] another possibility is to let CoAP end-points to notify their resources with the proposed */core-rd* interface. If each end-point is responsible to maintain the resource discovery up-to-date, the edge route will save many requests.

9.2.2 CoAP/HTTP mapping services

This subsection presents and discusses the main improvements offered by the CoAP/HTTP mapping services. These services provide interoperability with actual HTTP/1.1 clients and servers in order integrate WSNs in the actual Internet architecture.

CoAP/HTTP mapping services are divided in two groups:

1. **HTTP Subscriptions**. A Web browser can subscribe to CoAP servers using this service. Several method to reach this were analyzed and discussed in section 5.4. This particular implementation uses WebSockets to allow compatible Web clients to receive notifications from CoAP servers. With WebSockets only one request is needed to establish a bidirectional TCP connection between the Web browser and the CoAP proxy. In this case, the CoAP proxy sends all CoAP notifications back through the TCP connection only when a new notification arrives. In this way, the Web client does not need to resend a new HTTP request to get the next notification.

2. **Regular HTTP requests**. Web clients can send HTTP requests to the CoAP proxy in order to get a CoAP resource representation. These requests are handled by the proxy to transparently connect to CoAP servers and create a HTTP response. To analyze the latency introduced by the proxy in the HTTP/CoAP mapping, different tests have been performed using several payload values ranging from 10 to 1200 bytes. For each payload value, 100 latency measures have been done. Three test-bed scenarios have been used:

   - CoAP client embedded in a PC sending GET CON requests to a CoAP server embedded in a TelosB mote. In this case no CoAP/HTTP mapping was used.
   - HTTP client embedded in a PC sending GET requests to a HTTP/1.0 server embedded in a TelosB mote.
   - HTTP client embedded in a PC sending GET requests to a CoAP server embedded in a TelosB mote. This communication is handled by the CoAP proxy.

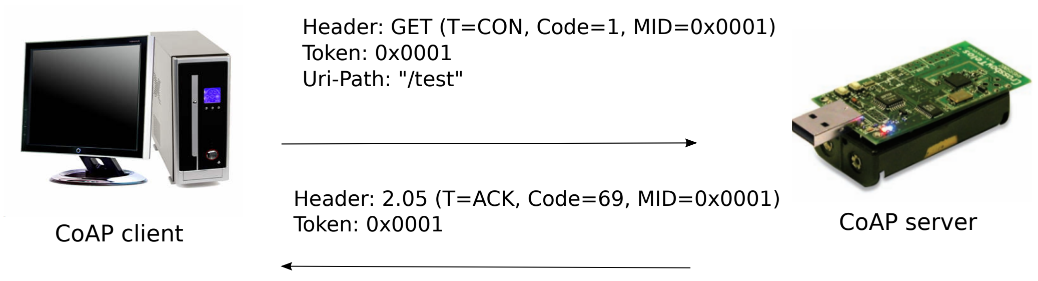

Header: GET (T=CON, Code=1, MID=0x0001)
Token: 0x0001
Uri-Path: "/test"
Header: 2.05 (T=ACK, Code=69, MID=0x0001)
Token: 0x0001
CoAP client
CoAP server

Figure 9.19: **Scenario 1**: CoAP client embedded in a PC sending GET CON requests to a CoAP server embedded in a TelosB mote.

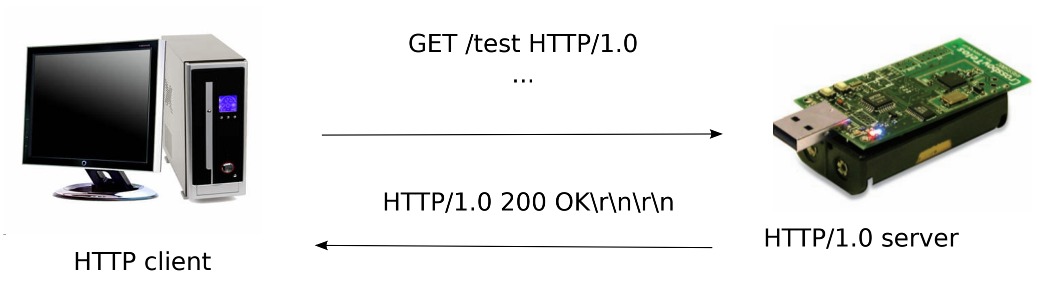


Figure 9.20: **Scenario 2**: HTTP client sending GET requests to a HTTP/1.0 server embedded in a TelosB mote.

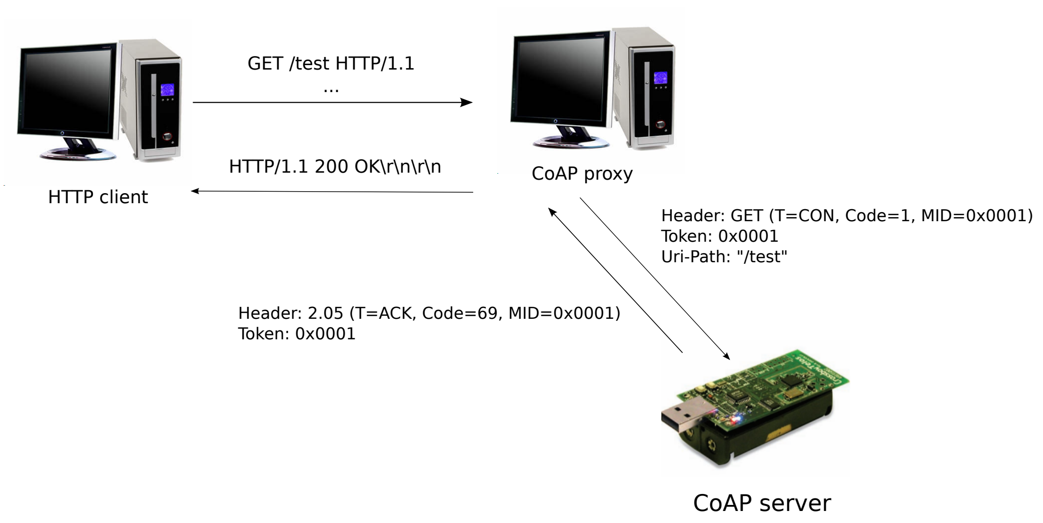


Figure 9.21: **Scenario 3**: HTTP client sending GET requests to a CoAP server embedded in a TelosB mote through the CoAP proxy.

Figure 9.22 shows the results obtained for the previous three scenarios.

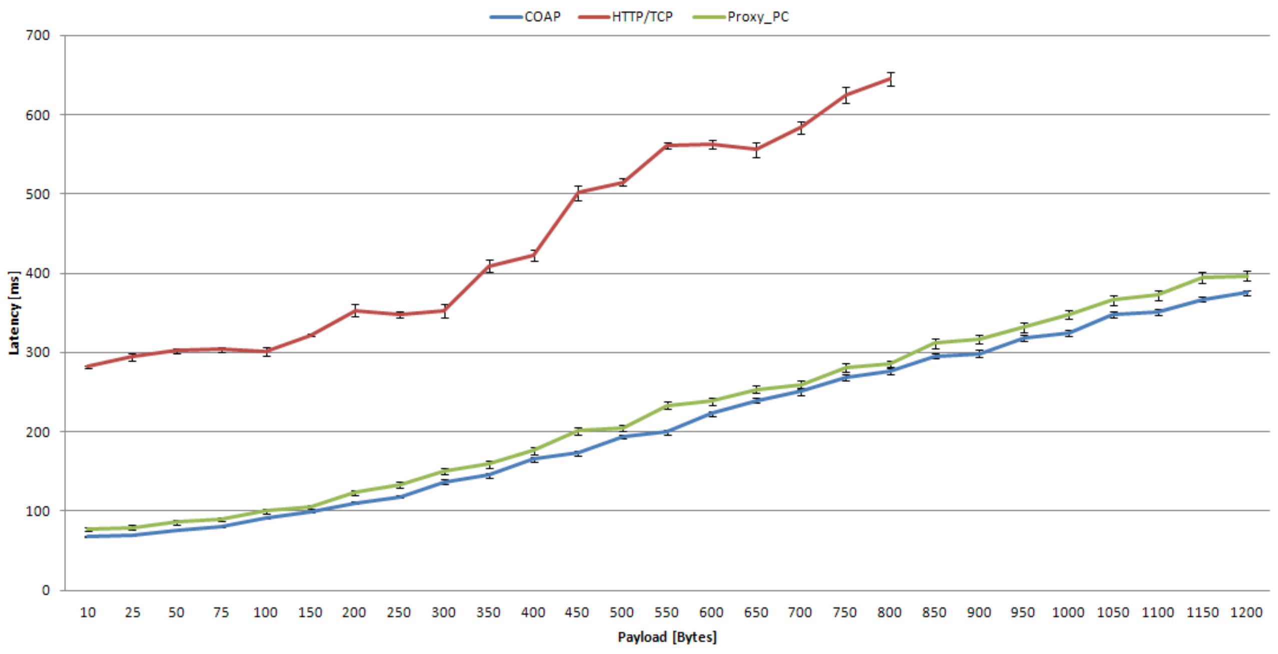


Figure 9.22: Performance evaluation for CoAP/HTTP translation

As results show, using a HTTP/1.0 server without using an intermediary produces high latency. As this can be the first alternative to allow Web clients to access sensor nodes, it is not a good alternative because of the high latency it produces.

Using a CoAP client directly to access a CoAP server has similar latency than using CoAP/HTTP mapping through the CoAP proxy. This extra latency introduced by the CoAP proxy is mainly due to protocol conversion. Using a CoAP client to obtain CoAP resources is a good choice and performs better than using an intermediary. However, using existing Web clients and services that use the HTTP protocol is a way to ensure interoperability. In cases where a CoAP client cannot be used, CoAP/HTTP mapping performed by the daemon performs similarly than using a CoAP client directly. Thus, using the CoAP proxy is a good choice in terms of performance to offer interoperability with Internet applications.

General conclusions will be presented in next subsection.

10 Conclusions and future work

10.1 Conclusions

This work motivates to use of Web services in WSNs. To overcome interoperability problems, the IEEE 802.15.4 protocol stack for physical and link layers and the 6LoWPAN protocol stack are presented, highlighting the constraints that WSNs networks have. Moreover, the CoAP application protocol is presented and proposed in order to provide a solution for upper layers in WSNs, which takes into account the main characteristics of these networks.

TinyOS operating system for constrained devices is also presented. The main aspects to take into account when creating new applications for this operating system have been explained, focusing on memory requirement that are crucial to create applications in devices with limited resources.

A software solution to integrate RESTful Web services in WSNs based on the CoAP protocol is proposed and developed. This software is a library for the TinyOS operating system that has been developed in order to easily create new applications that can use and offer Web-based services using the CoAP protocol. The hardware used to test this new library is TelosB.

In order to integrate the proposed sensor-based Web applications with the HTTP protocol, and therefore providing interoperability with the actual Internet, a solution has been proposed and developed. It is composed of a specific hardware and a new developed software. In order to create this software, a new framework for C applications that facilitates the integration of the CoAP protocol for the Linux operating system has been programmed. This new framework goes under the name of *CoAP tools* and provides the necessary tools to use and offer CoAP Web-based services in embedded devices with the Linux operating system.Using this framework, a new proxy for embedded devices has been programmed in order to allow external HTTP clients to connect to CoAP resources and offer other services to improve network throughput and expand 6LoWPAN network capabilities.

Finally, a performance evaluation has been done. In order to evaluate CoAP benefits, several test-bed scenarios have been used. Results show that using the HTTP in WSNs produces high latency in comparison with using the CoAP protocol. The main cause is that HTTP uses the TCP protocol that uses several messages to establish a TCP connection. In order to evaluate the CoAP implementation, it has been compared with using a HTTP over UDP server. In this case, transmission is not reliable because no extra logic is performed in the application layer by the server. Results show that using CoAP in constrained devices such as TelosB is a good alternative for WSNs because adds low latency in comparison with using a HTTP over UDP server. This low extra latency is justified because CoAP offers a simple reliability mechanism in order to perform better in scenarios where reliability is necessary among other features.

In order to evaluate the proposed proxy solution, several test-bed scenarios have been used. Results show that using the proxy solution adds low latency, thus providing a good solution when the use of a CoAP client is not possible.

10.2 Future work

In this subsection, some future tasks and considerations are described.

Due to some problems observed in the BLIP driver, performance benchmarking has not been possible in a different architecture than a PC because of payload limitations. The problem needs to be investigated in depth in order to know exactly the problems. It would be interesting to use another existing 6LoWPAN driver or a different interface in order to provide access to a 6LoWPAN network. It would be also interesting to integrate the optional draft for Blockwise transfers in CoAP[34] in *CoAP tools* and in TinyOS libraries. It can be interesting to enable media applications in WSNs.

Another interesting work is to evaluate and optimize the presented libraries taking into account power consumption and how to configure the libraries to work in several scenarios with different memory requirements.

Also programming the proposed TinyOS library in other operating systems for wireless sensor nodes would be interesting to compare their influence in the final performance results.

References

[1] IEEE Computer Society. *Wireless Medium Access Control (MAC) and Physical Layer (PHY). Specifications for Low-Rate Wireless Personal Area Networks (LR-WPANs)*, 2006. Part 15. 2006. Los Alamitos, CA, USA.

[2] G. Montenegro [et al.]. *Transmission of IPv6 Packets over IEEE 802.15.4 Networks.* Available online at: http://tools.ietf.org/search/rfc4944. September 2007 [Last access: 30 September 2011].

[3] P. Levis [et al.]. *TinyOS: An operating system for sensor networks.* in Ambient Intelligence, W. Weber, J. Rabaey, and E. Aarts, Eds. Berlin: Springer, 2005, pp. 115-148.

[4] T. Voigt A. Dunkels, B. Gröonvall. *Contiki - a lightweight and flexible operating system for tiny networked sensors.* In Proceedings of the First IEEE Workshop on Embedded Networked Sensors, Tampa, Florida, USA, November 2004.

[5] Crossbow Technology Inc. *TelosB Datasheet.* Available online at: http://www.willow.co.uk/TelosB_Datasheet.pdf [Last access: 30 September 2011].

[6] Z. Shelby [et al.]. *Constrained Application Protocol (CoAP), draft-ietf-core-coap-07.* Available online at http://tools.ietf.org/html/draft-ietf-core-coap-07. 8 July 2011 [Last access: 30 September 2011].

[7] C. Bormann Z. Shelby. *6LoWPAN: The Wireless Embedded Internet.* 1th ed.; John Wiley & Sons Ltd: Chichester, UK, 2009.

[8] *CoRE IETF Working Group.* Available online at: http://datatracker.ietf.org/wg/core/charter/ [Last access: 30 September 2011].

[9] S. Deering R. Hinden. *IP Version 6 Addressing Architecture.* Available online at: http://tools.ietf.org/html/rfc4291. February 2006. [Last access: 30 September 2011].

[10] Z. Shelby [et al.]. *6LoWPAN Neighbor Discovery. Internet-Draft draft-ietf-6lowpan-nd-06, Internet Engineering Task Force (work in progress).* Available online at: http://tools.ietf.org/html/rfc4291. September 2009. [Last access: 30 September 2011].

[11] F. Leymann C. Pautasso, O. Zimmermann. *RESTful Web Services vs. "Big" Web Services: Making the Right Architectural Decision.* WWW 2008 / Refereed Track: Web Engineering - Web Service Deployment. Beijing, China.

[12] M. Gudgin [et al.]. Soap version 1.2 part 1: Messaging framework (second edition), w3c recommendation. April 2007.

[13] R.T Fielding. *Architectural Styles and the Design of Network-based Software Architectures. PhD Thesis, University of California.* Irvine, 2000.

[14] W3C. *Extensible Markup Language (XML) 1.0 (Fifth Edition).* W3C Recommendation 26 November 2008 Available online at: http://www.w3.org/TR/REC-xml/. [Last access: 30 September 2011].

[15] W3C. *Web Services Description Language (WSDL) 1.1.* Web Services Description Language (WSDL) 1.1. Available online at: http://www.w3.org/TR/wsdl. [Last access: 30 September 2011].

[16] L. Clement [et al.]. *UDDI Version 3.0.2.* Available online at: http://www.uddi.org/pubs/uddi_v3.htm. 19 October 2004 [Last access: 30 September 2011].

[17] P. Levis. *TinyOS Programming.* Available online at: http://www.tinyos.net/tinyos-2.x/doc/pdf/tinyos-programming.pdf. 27 October 2006 [Last access: 30 September 2011].

[18] S. Krco Z. Shelby. *CoRE Resource Directory, draft-shelby-core-resource-directory-00.* Available online at: http://tools.ietf.org/html/draft-shelby-core-resource-directory-00. 27 June 211 [Last access: 30 September 2011].

[19] S. Krco Z. Shelby. *CoRE Resource Directory, draft-shelby-core-resource-directory-00.* The WWW Common Gateway Interface Version 1.1. Available online at: http://tools.ietf.org/html/draft-robinson-www-interface-00. 8 January 1996 [Last access: 30 September 2011].

[20] Inc. Open Market. *FastCGI: A High-Performance Web Server Interface.* Available online at: http://www.fastcgi.com/devkit/doc/fastcgi-whitepaper/fastcgi.htm. April 1996 [Last access: 30 September 2011].

[21] R. Fielding [et al.]. *Hypertext Transfer Protocol - HTTP/1.1.* RFC 2616, June 1999.

[22] P. McCarthy D. Crane. *Comet and Reverse Ajax: The Next Generation Ajax 2.0.* July 2008.

[23] S. Loreto [et al.]. *Known Issues and Best Practices for the Use of Long Polling and Streaming in Bidirectional HTTP.* Available online at: http://tools.ietf.org/html/rfc6202. April 2011 [Last access: 30 September 2011].

[24] I. Hickson. *The WebSocket protocol.* Available online at: http://tools.ietf.org/html/draft-hixie-thewebsocketprotocol-76. September 2010 [Last access: 30 September 2011].

[25] K. Kuladinithi [et al.]. *Implementation of CoAP and its Application in Transport Logistics.* Available online at: http://docs.tinyos.net/tinywiki/index.php/CoAP. 11 April 2011 [Last access: 30 September 2011].

[26] *BLIP tutorial.* Available online at: http://docs.tinyos.net/tinywiki/index.php/BLIP_Tutorial[Last access: 30 September 2011].

[27] D. Culler D. Gay, P. Levis. *Software Design Patterns for TinyOS.* In Proceedings of the 2005 ACM SIGPLAN/SIGBED Conference on Languages, Compilers, and Tools for Embedded Systems (LCTES), pages 40–49, New York, NY, USA, 2005. ACM Pres.

[28] Getting started with berkeley db. Available online at http://download.oracle.com/docs/cd/E17076_02/html/gsg/C/index.html. [Last access: 30 September 2011].

[29] Z. Shelby. *CoRE Link Format, draft-ietf-core-link-format-07.* Available online at: http://tools.ietf.org/html/draft-ietf-core-link-format-07. 25 July 2011 [Last access: 30 September 2011].

[30] Z. Shelby K. Hartke. *Observing Resources in CoAP, draft-ietf-core-observe-02*. Available online at: http://tools.ietf.org/html/draft-ietf-core-observe-02. 15 March 2011 [Last access: 30 September 2011].

[31] *Cambria GW2358-4 Network Computer*. Available online at: http://www.gateworks.com/products/cambria/datasheets/gw2358-4ds.pdf [Last access: 30 September 2011].

[32] *OpenWRT*. Available online at: https://openwrt.org. 11 November 2010 [Last access 30 September 2011].

[33] J.L Ferrer. *Instalación OpenWRT en las plataformas de Gateworks AVILA/CAMBRIA*. 11 November 2010.

[34] C. Bormann Z. Shelby. *Blockwise transfers in CoAP. Internet-Draft (work in progress)*. Available online at: http://tools.ietf.org/html/draft-ietf-core-block-04. 15 March 2011 [Last access: 30 September 2011].

List of Figures

List of Tables

Appendices

A CoAP TinyOS components

CoAP components are located in the *$TOSROOT/tos/lib/net/coap* directory. They are located in this folder because they are part of the operating system as they pretend to be reused between different applications.

CoAP resources are located in *$TOSROOT/apps/*custom_app directory. They are located in this folder because they are TinyOS applications written for specific purposes. Each application requires its own implementation, thus a new folder in *$TOSROOT/apps* should be created.

CoAP components are listed in the following list. All of them have two related files, one with a *C* a the end of the component name and one with a *P*. *C* files provide the interfaces and define the necessary wirings between components used by *P* files. *P* files contain all the logic written in nesC in order to provide the required functionality.

- *CoapPduC*. It is defined by two files: *CoapPduC.nc* and *CoapPduP.nc*. This component provides the *CoapPdu* interface that defines the necessary commands to handle CoAP PDUs (create, delete, modify options, etc...).
- *CoapOptionC*. It is defined by two files *CoapOptionC* and *CoapOptionP.nc*. This component provides the *CoapOption* interface that defines commands to create and delete CoAP options.
- *CoapServerC*. It is defined by two files: *CoapServerC.nc* and *CoapServerP.nc*. This component provides the *CoapServer* interface that defines the necessary commands to initialize a CoAP server.

CoAP resource components provide the *CoapResource* interface. It defines the necessary commands and events that every CoAP resource must have. When a new CoAP server is being implemented on a mote, CoAP resources must be programmed first. Programming a CoAP resource means creating a component that provides the *CoapResource* interface by programing its commands in nesC language. When a new resource is created, it needs to be wired to the CoapServerC component. This is done giving each resource a number and wiring it as a parametrized interface.

A.1 CoapOption interface

The following C structure is used by the CoapOption interface:

```
typedef struct {
    uint8_t code;
    uint16_t len;
```

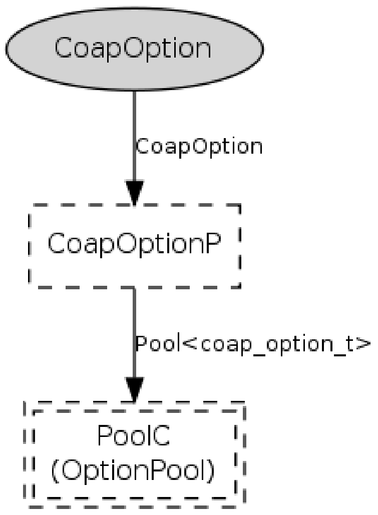


Figure A.23: CoapOption interface

```
    uint8_t data[MAX_OPT_DATA];
} coap_option_t;
```

It represents a CoAP option. It contains the option code (*code*), its value (*data*) and value's length (*len*).

CoapOption interface commands are the following:

```
command coap_option_t *create(uint8_t code, uint8_t *str, uint16_t len);
```

Creates a new `coap_option_t` structure getting the necessary memory from a PoolC configuration. It returns a pointer to a statically-allocated memory region that contains a `coap_option_t` structure. If fails, it returns a NULL pointer.

```
command void delete(coap_option_t *opt);
```

Frees statically-allocated memory used by the structure `coap_option_t`. Freeing in this context means returning the memory pointed by *opt* to the origin PoolC component.

A.2 CoapListC component

The following C structures are used by the CoapList interface:

```
struct coap_list_t {
    struct coap_node_t *first;
    struct coap_list_index_t  iterator;
};
```

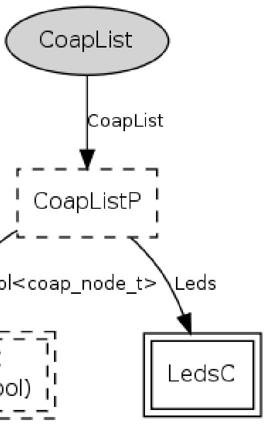


Figure A.24: CoapListC component

It represents a list of elements. It has a pointer to a linked list C structure which contains pointers to `coap_node_t` structures that contain any type of data. It also contains an iterator to easily iterate through list elements. C *coap_list_t* structure is defined as follows:

```
struct coap_node_t {
    struct coap_node_t *next;
    uint8_t key;
    void *data;
};
```

It represents a list element. It has a pointer to the next element in the linked list (*next*), an integer key to identify the element (*key*) and a void pointer to the element's data (*data*).

List iterator is defined by the following C structure:

```
struct coap_list_index_t {
    struct coap_list_t *list;
    struct coap_node_t *this, *next;
};
```

Each iterator contains a pointer to the origin list and pointer to the current element and the next element.

CoapList interface has the following commands:

```
command void initList(coap_list_t *list);
```

Initializes a *coap_list_t* structure. It needs to be called before working with any list.

```
command coap_node_t *createListNode(uint8_t key, void *data);
```

Creates a new node getting the necessary memory from a PoolC configuration. It returns a pointer to a statically-allocated memory region that contains a *coap_node_t structure*. If fails, it returns a NULL pointer.

```
command error_t insertListNode(coap_list_t *list, uint8_t key, void *data);
```

Inserts a new list element. The new element is created and allocated using the *key* and *data* parameters. Returns SUCCESS if the element can be inserted and FAIL if not.

```
command error_t getListNode(coap_list_t *list, uint8_t key, void **data);
```

Gets list element's data given the element's key. A pointer to the obtained element is returned in the *data* parameter. Returns SUCCESS if the element is found and FAIL if not.

```
command error_t deleteListNode(coap_list_t *list, int key, void **data);
```

Deletes a list element given the element's key. A pointer to the deleted element is returned in the *data* parameter. It returns SUCCESS if the element can be deleted and FAIL if not.

```
command coap_list_index_t *first(coap_list_t *list);
```

Returns an iterator with the first element in the list. Returns NULL if no element is found.

```
command coap_list_index_t *next(coap_list_index_t *li);
```

Returns an iterator with the next element in list. Returns NULL if no node is found.

```
command void this(coap_list_index_t *li, uint8_t *key, void **data);
```

Returns list element's key and data from an iterator.

A.3 CoapPdu interface

The following C structures are used by the CoapPdu interface:

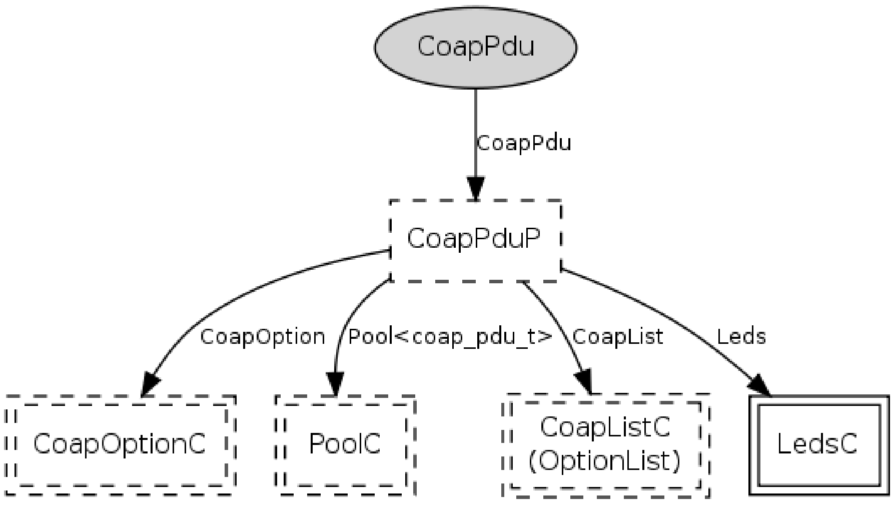


Figure A.25: CoapPdu interface

```
typedef struct {
    uint8_t timestamp;
    coap_hdr_t hdr;
    struct sockaddr_in6 source_addr;
    coap_list_t opt_list;
} coap_pdu_t;
```

It represents a CoAP PDU. It contains a time stamp (*timestamp*) to know when the PDU was created, a header (*hdr*) containg all header information, the source address (*source_addr*) and a list of CoAP options (*opt_list*). The header struct is the following:

```
typedef struct {
    uint8_t version;
    uint8_t type;
    uint8_t code;
    uint8_t optcnt;
    uint16_t id;
} coap_hdr_t;
```

It contains the protocol version (*version*), the message type (*type*), the response or request code (*code*), the number of options (*optcnt*) and the message ID (*id*) All these fields are detailed in section 3.2.2.

CoapPdu interface commands are:

```
command coap_pdu_t *create();
```

Creates a new `coap_pdu_t` struct getting the necessary memory from a PoolC component. It returns a pointer to a statically-allocated memory region that contains a `coap_pdu_t` structure. If fails, it returns a NULL pointer.

```
command void delete(coap_pdu_t *pdu);
```

Frees statically-allocated memory used by the structure `coap_pdu_t` and by its options. Freeing in this context means returning the memory pointed by *pdu* to the origin PoolC component.

```
command void clean(coap_pdu_t *pdu);
```

Frees statically-allocated memory used by PDU options and clears the `coap_pdu_t` structure.

```
command error_t packetRead(coap_pdu_t *pdu, uint8_t *buffer, uint16_t packet_len,
        uint8_t *payload, uint16_t *payload_len);
```

Parses a CoAP packet which is pointed by *buffer* which length is *packet_len* and fills a previously created `coap_pdu_t` structure (*pdu*) with its content. It also returns the PDU payload in the *payload* buffer provided and its length (*payload_len*). Returns SUCCESS if succeeds and FAIL if fails.

```
command error_t packetWrite(coap_pdu_t *pdu, uint8_t *buffer, uint8_t *payload,
        uint16_t payload_len, uint16_t *packet_len);
```

Creates a CoAP packet into the buffer pointed by *buffer* given a CoAP PDU (*pdu*) and a payload (*payload*) with length *payload_len*. It returns a buffer with the new packet created and its length (*packet_len*). Returns SUCCESS if succeeds and FAIL if fails.

```
command error_t insertOption(coap_pdu_t *pdu, uint8_t code, uint8_t *str, uint16_t len);
```

Inserts a new CoAP option into the CoAP PDU pointed by *pdu* structure given the option's code (*code*), value (*str*) and length (*len*). Returns SUCCESS on success and FAIL on failure. Several predefined constants are available in the *coap_option.h* header:

```
COAP_OPTION_CONTENT_TYPE
COAP_OPTION_MAX_AGE
COAP_OPTION_PROXY_URI
COAP_OPTION_ETAG
COAP_OPTION_URI_HOST
COAP_OPTION_LOCATION_PATH
COAP_OPTION_URI_PORT
COAP_OPTION_LOCATION_QUERY
COAP_OPTION_URI_PATH
COAP_OPTION_TOKEN
COAP_OPTION_URI_QUERY
```

```
command error_t getOption(coap_pdu_t *pdu, uint8_t code, coap_option_t **opt);
```

Gets a pointer to `coap_option_t` structure identified by an option's code (*code*) from the CoAP PDU (*pdu*). Returns SUCCESS if the option is found and FAIL if not.

```
command error_t unsetOption(coap_pdu_t *pdu, uint8_t code);
```

Deletes and frees an option given the option's code (*code*) from a CoAP PDU (*pdu*). Returns SUCCESS if the option can be deleted and FAIL if not.

```
command error_t checkOptions(coap_pdu_t *pdu, coap_option_t **option);
```

Checks for critical unrecognized options and, if found, an option parameter (*option*) is returned with a pointer to the first unrecognized option found. Returns SUCCESS if no unrecognized options are found and FAIL if some unrecognized option is found.

A.4 CoapServer interface

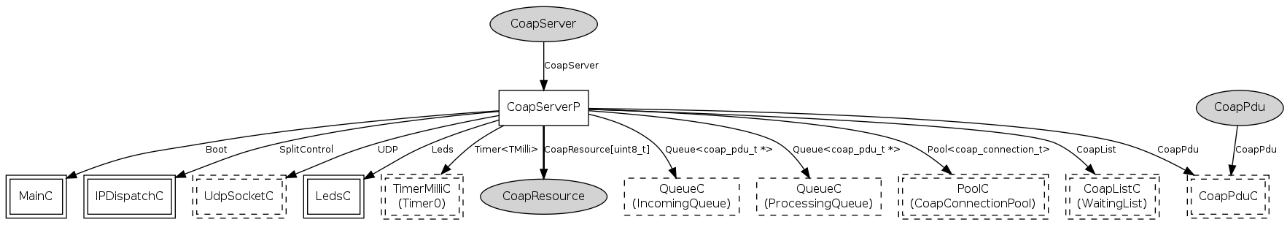


Figure A.26: CoapServer interface

The following C structure is used by the CoapServer interface:

```
typedef struct {
    coap_pdu_t *pdu;
    uint8_t payload[PAYLOAD_SIZE];
    uint16_t payload_len;
    uint8_t nretransmit;
} coap_connection_t;
```

It represents a new CoAP connection. It contains a pointer to a CoAP PDU structure (*pdu*), the PDU payload (*payload*) and its length (*payload_len*) and the number of times it has been retransmitted (*nretransmit*).

The following commands and events are defined by the CoapServer interface:

```
command error_t init(uint16_t port, key_uri_t *map, uint8_t map_len);
```

Initializes the CoAP server in the port *port* with *map_len* resources given by the *map* parameter. It returns SUCCESS if the server has been correctly initialized and FAIL if not.

```
event void booted();
```

This event is signaled when the CoAP server has been corretly started.

A.5 CoapResource interface

This interface can used by TinyOS applications. It identifies a CoAP resource that can be accessed using CoAP messages. It uses the following C structure:

```
typedef struct key_uri {
    uint8_t key;
    uint8_t uri[MAX_URI_LEN];
    uint8_t len;
    uint8_t rt[10];
    uint8_t iff[10];
    uint8_t sz;
    uint8_t ct;
  } key_uri_t;
```

The *key* field identifies the resources and it is used to identify the resource as a parametrized interface. *uri* contains the CoAP URI to access the resource which length is *len*. Fields *rt*, *iff*, *sz* and *ct* correspond to Resource type, Interface Description, Size and Content-Type CoRE Link format properties reviewed in section 3.3.

The following commands and events are used by the CoapResource interface:

```
command void handle(coap_connection_t *conn);
```

Defines the logic for handle a CoAP connection.

```
event void isSeparateResponse(coap_connection_t *conn);
```

This event is signaled when a separate response is needed. The current connection must be provided.

```
event void isDone(coap_pdu_t *response, uint8_t is_subscription);
```

This event is signaled when the connection processing has been done. It must provide the current connection with a CoAP PDU response which is specified by the *pdu* and *payload* fields of the *conn* structure. *is_subscription* is used to indicate if the response is a notification (1) or not (0).

A.6 CoapClient interface

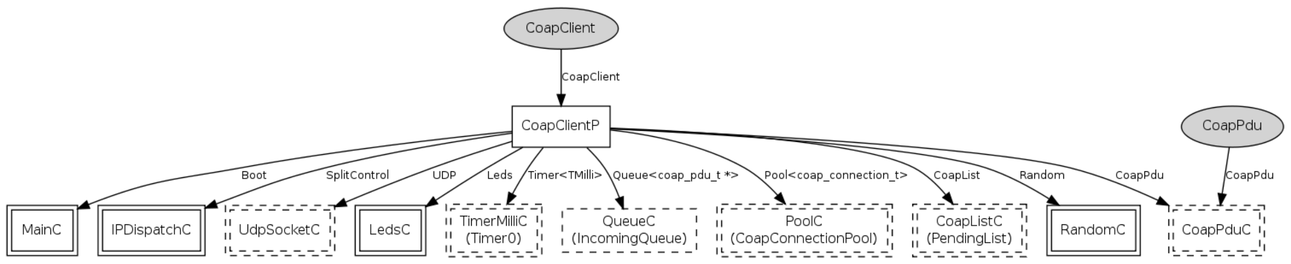


Figure A.27: CoapClient interface

It represents a new CoAP client. This interface can be used to send and receive CoAP messages.

The following commands and events are defined by the CoapClient interface:

```
command void sendRequest(coap_pdu_t *request);
```

Sends a new CoAP message.

```
event void messageReceived(coap_pdu_t *response);
```

This event is signaled when a new response is received.

B Existing software cross-compilation and installation

Constants used in this appendix:

CC_HOME: the path where the *target-armeb_v5te_uClibc-0.9.30.3* is located.

TOS_HOME: the path where TinyOS is located.

B.1 BLIP driver cross compilation

This steps have been used to cross compile the BLIP driver in the *GW2358-4* network computer:

1. Open a new shell.
2. Export the CC environment variable with CC_HOME value:

   ```
   export CC=$CC_HOME
   ```

3. Go to TOS_HOME/support/sdk/c/sf and issue the following commands:

   ```
   cd $TOS_HOME/support/sdk/c/sf
   ./configure --host=arm-linux
   make
   make install
   ```

4. The BLIP driver has a small issue in the *GW2358-4* network computer. It is not detecting new devices connecting to the network. In order to solve this, put the function *add_report* out of the `switch` in the file *serial_tun.conf* in the function *process_dest_tlv*.
5. Go to TOS_HOME/support/sdk/c/blip and issue the following commands:

   ```
   cd $TOS_HOME/support/sdk/c/blip
   ./configure --host=arm-linux
   make
   make install
   ```

6. Copy the created *driver/ip-driver* executable to the server (for instance into /www folder):

   ```
   scp driver/ip-driver root@192.168.3.23:/www
   ```

B.2 Lighttpd Web server configuration

```
server.document-root = "/var/www/pfc/public"

server.port = 3000

mimetype.assign = (
  ".html" => "text/html",
  ".txt" => "text/plain",
  ".jpg" => "image/jpeg",
  ".png" => "image/png",
  ".css" => "text/css",
  ".js" => "text/javascript"
)

server.modules = (
"mod_fastcgi",
"mod_rewrite",
)
url.rewrite-once = ( "^/([a-zA-Z0-9_]+)$" => "/coap.do?path=$1")
fastcgi.server = (
    "coap.do" => ((
    "socket" => "/tmp/coap_fastcgi",
    "check-local" => "disable"
    )),
    ".php" => ((
         "bin-path" => "/opt/php/bin/php-cgi",
         "socket" => "/tmp/php.socket"
    ))
)
```

In this particular example, the Web server is running in the port 3000 and all requests are redirected to the path *coap.do* and the requests is passed in the GET parameter *path*. This request is mapped to a FastCGI process through the UNIX socket */tmp/coap_fastcgi*. Another FastCGI process is listening in the UNIX socket */opt/php/bin/php-cgi* in order to enable PHP applications.

C CoAP tools API

C.1 coap_net.h

Common network functions.

C.1.1 Functions

```
int coap_net_address_create(struct sockaddr_in6 *addr, char *hostname, int port);
```

Creates a new IPv6 address given a host name and a port. Returns 1 on success or 0 on failure.

Parameters:

- *addr*. The IPv6 address to return.
- *hostname*. The host name.
- *port*. The port.

```
int coap_net_udp_create();
```

Creates a new UDP socket. Returns the socket number on success or -1 on failure.

```
int coap_net_udp_bind(int port);
```

Creates and binds an UDP socket in the specified port. Returns the socket number on success or -1 on failure.

Parameters:

- *port*. The port to bind.

```
int coap_net_tcp_connect(char *hostname,int port);
```

Creates a new TCP socket to connect to a host name using a specified port. Returns the socket number on success or -1 on failure.

Parameters:

- *hostname*. The host name.
- *port*. Port.

```
int coap_net_tcp_bind(int port);
```

Creates and binds a new TCP socket in the specified port. Returns the socket number on success or -1 on failure.

Parameters:

- *port*. The port.

C.2 coap_list.h

Encapsulates a linked list.

C.2.1 Data structures

```
struct coap_node_t {
    struct coap_node_t *next;
    char *key;
    void *data;
};

struct coap_list_index_t {
    struct coap_list_t *list;
    struct coap_node_t *this, *next;
};
```

```
struct coap_list_t {
    struct coap_node_t *first;
    struct coap_list_index_t  iterator;  /* For apr_hash_first(NULL, ...) */
    void (*delete_func)(struct coap_node_t *);
    int (*order_func)(void *, void *);
};
```

C.2.2 Functions

```
int coap_list_create(void (*delete_func)(struct coap_node_t *),
         int (*order_func)(void *, void *), coap_list_t **list);
```

Creates a new linked list.

Parameters:

- *delete_func.* A function called when a node is removed from the list.
- *order_func.* A function used to compare two data structures.
- *list.* The list to allocate from.

```
int coap_list_node_insert(coap_list_t *list, void *key, int len, void *data);
```

Inserts a new node into the list. Returns 1 on success or 0 on failure.

Parameters:

- *list.* The list to modify.
- *key.* The key that identifies the node.
- *len.* The length of the key.
- *data.* The data to insert into the node.

`int coap_list_node_get(coap_list_t *list, void *key, int len, void **data);`

Gets node's data. Returns 1 on success or 0 on failure.

Parameters:

- *list.* The list used to get node's information.
- *key.* The key that identifies the node.
- *len.* The length of the key.
- *data.* Node's data memory region.

`int coap_list_node_create(void *key, int len, void *data, coap_node_t **node);`

Creates a new node. Returns 1 on success or 0 on failure.

Parameters:

- *key.* The key to identify the node.
- *len.* The length of the key.
- *data.* Node's data.
- *len.* The node to allocate from.

`int coap_list_node_delete(coap_list_t *list, void *key, int len);`

Removes a node from the list. Returns 1 on success or 0 on failure.

Parameters:

- *list.* The list to modify.
- *key.* The key that identifies the node to be removed.
- *len.* The length of the key.

`int coap_list_clean(coap_list_t *list);`

Cleans all list nodes. It does not free the memory used by the list. Returns 1 on success or 0 on failure.

Parameters:

- *list.* The list to clean.

`int coap_list_delete(coap_list_t *list);`

Removes all list nodes and frees the list. Returns 1 on success or 0 on failure.

Parameters:

- *list.* The list to delete.

`coap_list_index_t *coap_list_first(coap_list_t *list);`

Returns the first iteration state from a list. Returns NULL if the list is empty.

Parameters:

- *list.* The list to be iterated.

`coap_list_index_t *coap_list_next(coap_list_index_t *li);`

Continue iterating over the entries in a list. Returns NULL if next node is the end of the list.

Parameters:

- *li.* Iteration state.

```
void coap_list_this(coap_list_index_t *li, const void **key,void **data);
```

Get the current node's details from the iteration state

Parameters:

- *li.* Iteration state.
- *key.* Return pointer for the pointer to the key.
- *data.* Return pointer for the associated data.

C.3 coap_general.h

```
#define RESPONSE_TIMEOUT 2
#define MAX_RETRANSMIT 4
#define COAP_DEFAULT_RESPONSE_TIMEOUT  5
#define COAP_DEFAULT_MAX_RETRANSMIT    5
#define COAP_DEFAULT_PORT          5683
#define COAP_DEFAULT_MAX_AGE          60
#define COAP_MAX_PDU_SIZE           1400
#define COAP_MAX_PAYLOAD_SIZE           1400
#define COAP_DEFAULT_VERSION           1
#define COAP_DEFAULT_SCHEME        "coap"
#define COAP_DEFAULT_URI_WELLKNOWN ".well-known/core"
#define COAP_MEDIATYPE_TEXT_PLAIN                        0
#define COAP_MEDIATYPE_LINK_FORMAT                       40
#define COAP_MEDIATYPE_XML                               41
#define COAP_MEDIATYPE_OCTET_STREAM                      42
#define COAP_MEDIATYPE_EXI                               47
#define COAP_MEDIATYPE_JSON                              50
```

C.4 coap_context.h

C.4.1 Data structures

```
struct coap_context_t {
        int socket;
        coap_list_t *pending_queue;
        coap_list_t *dispatch_queue;
        unsigned int message_id;
        unsigned int token;
};
```

```
typedef struct {
      coap_pdu_t *pdu;
      unsigned int n_retransmit;
      time_t timestamp;
      unsigned int is_subscription;
      unsigned int is_separate_response;
      time_t last_valid_notification;
      time_t last_observe_value;
} coap_context_connection_t;
```

C.4.2 Functions

```
int coap_context_create(coap_context_t **context);
```

Creates a new context. Returns 1 on success or 0 on failure.

Parameters:

- *context.* Return pointer for the pointer to the allocated context.

```
int coap_context_connection_create(coap_pdu_t *pdu,
         coap_context_connection_t **conn);
```

Creates a new connection. Returns 1 on success or 0 on failure.

Parameters:

- *pdu.* The CoAP PDU to insert into the connection.
- *conn.* Return pointer for the pointer to the allocated connection.

```
void coap_context_connection_node_delete(struct coap_node_t *node);
```

Handler to remove a connection node from a list.

Parameters:

- *node.* The list node.

```
void coap_context_delete(coap_context_t *context);
```

Frees a context.

Parameters:

- *context.* The context to remove.

```
void coap_context_read(coap_context_t *context);
```

Reads incoming packets from the context socket.

Parameters:

- *context.* The context to read incoming packets.

```
void coap_context_send(coap_context_t *context,
coap_pdu_t *pdu);
```

Sends a CoAP PDU using the context socket.

Parameters:

- *context.* The context that contains the socket to send the CoAP PDU.
- *pdu.* The CoAP PDU to send.

C.5 coap_option.h

C.5.1 Data structures

```
#define MAX_OPT_DATA 270
typedef struct {
        char code;
        int len;
        char data[MAX_OPT_DATA];
} coap_option_t;

enum {
        COAP_OPTION_CONTENT_TYPE = 1,
        COAP_OPTION_MAXAGE = 2,
        COAP_OPTION_PROXY_URI = 3,
        COAP_OPTION_ETAG = 4,
        COAP_OPTION_URI_AUTHORITY = 5,
        COAP_OPTION_LOCATION = 6,
        COAP_OPTION_URI_PATH = 9,
        COAP_OPTION_TOKEN = 11,
        COAP_OPTION_URI_QUERY = 15,
        COAP_OPTION_OBSERVE = 17
};
```

C.5.2 Functions

```
int coap_option_create(char code, void *value, int len, coap_option_t **opt);
```

Creates a new option. Returns 1 on success or 0 on failure.

Parameters:

- *code.* The CoAP option code.
- *value.* The CoAP option value.
- *len.* The CoAP option value length.
- *opt.* Return pointer for the pointer to the allocated CoAP option.

```
int coap_option_delete(coap_option_t *opt);
```

Frees a CoAP option. Returns 1 on success or 0 on failure.

Parameters:

- *opt*. The CoAP option to remove.

C.6 coap_pdu.h

C.6.1 Data structures

```
enum {
        COAP_MESSAGE_CON = 0,
        COAP_MESSAGE_NON = 1,
        COAP_MESSAGE_ACK = 2,
        COAP_MESSAGE_RST = 3
};
enum {
        COAP_REQUEST_GET = 1,
        COAP_REQUEST_POST = 2,
        COAP_REQUEST_PUT = 3,
        COAP_REQUEST_DELETE = 4
};

enum {
       COAP_RESPONSE_201 = 65,
       COAP_RESPONSE_202 = 66,
       COAP_RESPONSE_203 = 67,
       COAP_RESPONSE_204 = 68,
       COAP_RESPONSE_205 = 69,
       COAP_RESPONSE_400 = 128,
       COAP_RESPONSE_401 = 129,
       COAP_RESPONSE_402 = 130,
       COAP_RESPONSE_403 = 131,
       COAP_RESPONSE_404 = 132,
       COAP_RESPONSE_405 = 133,
       COAP_RESPONSE_413 = 141,
       COAP_RESPONSE_415 = 143,
       COAP_RESPONSE_500 = 160,
       COAP_RESPONSE_501 = 161,
       COAP_RESPONSE_502 = 162,
       COAP_RESPONSE_503 = 163,
       COAP_RESPONSE_504 = 164,
       COAP_RESPONSE_505 = 165
};

typedef struct {
      char version;
      unsigned char type;
      unsigned char code;
      unsigned char optcnt;
      unsigned short id;
} coap_hdr_t;

typedef struct {
      coap_hdr_t hdr;
      coap_list_t *opt_list;
      char payload[COAP_MAX_PAYLOAD_SIZE];
      unsigned int payload_len;
      struct sockaddr_in6 addr;
      time_t timestamp;
} coap_pdu_t;
```

C.6.2 Functions

```
int coap_pdu_create(coap_pdu_t **pdu);
```

Creates a new CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *pdu*. Return pointer for the pointer to the allocated CoAP PDU.

```
int coap_pdu_delete(coap_pdu_t *pdu);
```

Frees a CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *pdu*. The CoAP PDU to remove.

```
void coap_pdu_node_delete(struct coap_node_t *node);
```

Function callback to remove a CoAP PDU from a list.

Parameters:

- *node*. The node in the list.

```
int coap_pdu_clean(coap_pdu_t *pdu);
```

Cleans a CoAP PDU. It does not free the CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *pdu*. The CoAP PDU to clean.

```
int coap_pdu_copy(coap_pdu_t *old, coap_pdu_t *new);
```

Copies the content of one CoAP PDU to another.

Parameters:

- *old*. The CoAP PDU to be copied..
- *new*. The new CoAP PDU where to copy the old one.

```
int coap_pdu_option_insert(coap_pdu_t *pdu, char code, char *value, unsigned int len);
```

Inserts a new CoAP option into the CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *pdu*. The CoAP PDU to be modified.
- *code*. The CoAP option code.
- *value*. The CoAP option value.
- *len*. The CoAP option value length.

```
int coap_pdu_option_delete(coap_pdu_t *pdu);
```

Frees a CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *pdu.* The CoAP PDU to be freed.

```
int coap_pdu_option_get(coap_pdu_t *pdu, char code, coap_option_t **opt);
```

Gets a CoAP option from a CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *pdu.* The CoAP PDU.
- *code.* The CoAP option code.
- *opt.* Return pointer for the pointer to the CoAP option.

```
int coap_pdu_option_unset(coap_pdu_t *pdu, char code);
```

Removes a CoAP option from a CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *pdu.* The CoAP PDU.
- *code.* The CoAP option code of the option to be removed.

```
int coap_pdu_packet_write(coap_pdu_t *pdu, char *buffer,  unsigned int *packet_len);
```

Writes a CoAP PDU in a buffer ready to be sent through a socket. Returns 1 on success or 0 on failure.

Parameters:

- *pdu*. The CoAP PDU.
- *buffer*. The buffer to contain the CoAP packet.
- *len*. Return pointer for the CoAP packet length.

```
int coap_pdu_packet_read(char *buffer, unsigned int packet_len, coap_pdu_t *pdu);
```

Reads a CoAP packet from a buffer and creates a new CoAP PDU. Returns 1 on success or 0 on failure.

Parameters:

- *buffer*. The buffer containing the CoAP packet.
- *len*. CoAP packet length.
- *pdu*. Return pointer for the CoAP PDU containing the CoAP packet.

```
void coap_pdu_print(coap_pdu_t *pdu);
```

Prints CoAP PDU fields using the standard error.

Parameters:

- *pdu*. The CoAP PDU to print.

```
int coap_pdu_parse_uri(char *coap_uri, struct sockaddr_in6 *addr,
        char **uri_path, char **uri_query);
```

Given a CoAP URI returns the server address, the URI path and the query string. Returns 1 on success or 0 on failure.

Parameters:

- *coap_uri*. The CoAP URI.
- *addr*. Return pointer to the server address.
- *uri_path*. Return pointer for the pointer to the URI path.
- *uri_query*. Return pointer for the pointer to the query string.

```
int coap_pdu_parse_query(char *query, char *variable, int len, char **value);
```

Parses a query string, filtering a desired GET parameter. Returns 1 on success or 0 on failure.

Parameters:

- *query*. The query string.
- *variable*. The GET parameter to filter.
- *len*. Length of the GET parameter specified.
- *value*. Return pointer for the pointer to the query string value of the GET parameter.

C.7 coap_server.h

C.7.1 Data structures

```
enum {
        RD_INSERT,
        RD_UPDATE,
        RD_DELETE
};

typedef struct {
        coap_resource_t *resource;
        coap_list_t *observers;
        unsigned int observe;
        time_t last_timestamp;
} coap_server_subscription_t;

typedef struct {
        coap_context_t *context;
        coap_list_t *resources;
        coap_list_t *subscriptions;
        coap_list_t *proxy;
        coap_rd_t *rd;
        int (*rd_update)(int, coap_rd_entry_t *, coap_resource_t *);
        coap_client_t *client;
} coap_server_t ;
```

C.7.2 Functions

```
int coap_server_create(int port, int cache, coap_server_t **server);
```

Creates a new CoAP server. Returns 1 on success or 0 on failure.

Parameters:

- *port.* The port to listen on.
- *cache.* Use caching (1) or not (0).
- *server.* Return pointer for the pointer to the allocated CoAP server.

```
void coap_server_delete(coap_server_t *server);
```

Frees a CoAP server.

Parameters:

- *server.* The CoAP server to be freed.

```
void coap_server_read(coap_server_t *server);
```

Reads an incoming packet from the server socket.

Parameters:

- *server.* The CoAP server that contains the socket to receive the packet.

```
void coap_server_request_send(coap_server_t *server, coap_pdu_t *pdu);
```

Sends a CoAP PDU using the server socket.

Parameters:

- *server.* The CoAP server.
- *pdu.* The CoAP PDU to be sent.

```
void coap_server_response_send(coap_server_t *server, coap_pdu_t *pdu);
```

Sends CoAP PDU response using the server socket.

Parameters:

- *server.* The CoAP server.
- *pdu.* The CoAP PDU to be sent.

```
void coap_server_dispatch(coap_server_t *server);
```

Dispatches received packets.

Parameters:

- *server*. The CoAP server to dispatch packets.

```
void proxy_dispatch(coap_server_t *server);
```

Dispatches packets coming from a proxy connection.

Parameters:

- *server*. The CoAP server to dispatch packets.

```
int coap_server_resource_add(coap_server_t *server, coap_resource_t *resource);
```

Adds a new resource to the CoAP server. Returns 1 on success or 0 on failure.

Parameters:

- *server*. The CoAP server.
- *resource*. The resource to be added.

```
void coap_server_notify_observers(coap_server_t *server, coap_resource_t *res,
         coap_pdu_t *response);
```

Notifies a list of observers with a CoAP PDU response.

Parameters:

- *server*. The CoAP server that sends the packets.
- *res*. The resource with the list of observers.

- *response.* The CoAP PDU response.

```
void coap_server_clean_pending(coap_server_t *server);
```

Cleans the list of pending CoAP packets. It includes retransmissions and removal of the expired connections.

Parameters:

- *server.* The CoAP server.

```
int coap_server_rd_update(coap_server_t *server, coap_rd_entry_t *rd,
        char *data, int len);
```

Updates the CoAP resource directory. Returns 1 on success or 0 on failure.

Parameters:

- *server.* The CoAP server.
- *rd.* The new entry of the resource directory.
- *data.* Name for the entry.
- *len.* Entry's name length.

C.8 coap_client.h

C.8.1 Data structures

```
typedef struct {
         coap_context_t *context;
         void (*response_received)(coap_pdu_t *, void *);
         void (*pending_removed)(char *, void *);
         DB *cache_db;
         void *data;
} coap_client_t ;
```

C.8.2 Functions

```
int coap_client_create(DB *cache_db,
        void *data,
        void (*response_received)(coap_pdu_t *, void *),
        void (*request_removed)(char *, void *),
        coap_client_t **client);
```

Creates a new CoAP client. Returns 1 on success or 0 on failure.

Parameters:

- *cache_db*. The database to store cache entries.
- *data*. Optional parameter that will be passed to function callbacks.
- *response_received*. Function callback called when a new CoAP response is received.
- *request_removed*. Function callback called when a CoAP request is removed.
- *client*. The CoAP client.

```
void coap_client_delete(coap_client_t *client);
```

Frees a CoAP client.

Parameters:

- *client*. The CoAP client.

```
void coap_client_read(coap_client_t *client);
```

Reads incoming packets.

Parameters:

- *client*. The CoAP client.

```
int coap_client_send(coap_client_t *client,
coap_pdu_t *request,
int subscription);
```

Sends a CoAP request. Returns 1 on success or 0 on failure.

Parameters:

- *client.* The CoAP client.
- *request.* The CoAP PDU request
- *subscription.* Indicates if the request is a subscription (1) or not (0).

```
void coap_client_dispatch(coap_client_t *client);
```

Dispatches received packets.

Parameters:

- *client.* The CoAP client.

```
void coap_client_clean_pending(coap_client_t *client);
```

Cleans pending packets. It includes doing the necessary retransmissions.

Parameters:

- *client.* The CoAP client.

C.9 coap_cache.h

C.9.1 Data structures

```
typedef struct {
        time_t timestamp;
        char packet[COAP_MAX_PDU_SIZE];
        int packet_len;
        struct sockaddr_in6 addr;
} coap_cache_t;
```

C.9.2 Functions

```
void coap_hash(coap_pdu_t *pdu, char *digest);
```

Creates a new CoAP hash.

Parameters:

- *pdu.* The CoAP PDU to perform the hash on.
- *disgest.* Return pointer to the CoAP PDU hash.

```
int coap_cache_create(char *database_path, DB **database);
```

Creates a new database. Returns 1 on success or 0 on failure.

Parameters:

- *database_path.* The path to store the database file.
- *database.* Return pointer for the pointer to the allocated database.

```
int coap_cache_add(DB *database, coap_pdu_t *request, coap_pdu_t *response);
```

Adds a new response to the cache database. Returns 1 on success or 0 on failure.

Parameters:

- *database.* The database to store the cache.
- *request.* The request that received the response.
- *response.* The response to store.

```
int coap_cache_get(DB *database, coap_pdu_t *request, coap_pdu_t **pdu);
```

Gets a stored response from the cache database. Returns 1 on success or 0 on failure.

Parameters:

- *database.* The database search for the cache.
- *request.* The request to identify the cache.
- *pdu.* Return pointer for the pointer to the allocated CoAP PDU.

```
int coap_cache_revalidate(DB *database, coap_pdu_t *request, coap_pdu_t **response);
```

Revalidates a cache if it is not fresh. Returns 1 on success or 0 on failure.

Parameters:

- *database.* The database search for the cache.
- *request.* The request to identify the cache.
- *response.* Return pointer for the pointer to the stored response.

```
int coap_cache_delete(DB *database, coap_pdu_t *pdu);
```

Removes a cache from the cache database. Returns 1 on success or 0 on failure.

Parameters:

- *database.* The database search for the cache.
- *pdu.* The request to identify the cache.

C.10 coap_resource.h

C.10.1 Data structures

```
struct coap_resource_t {
        char *path;
        char *name;
        char *rt;
        char *ifd;
        int sz;
        struct sockaddr_in6 server_addr;
        unsigned int is_subscription;
        void (*handler)(coap_pdu_t *);
};
```

C.10.2 Functions

```
int coap_resource_create(char *path, char *name, char *rt, char *ifd, int sz,
        int is_subscription, void (*handler)(coap_pdu_t *),
        coap_resource_t **resource);
```

Creates a new resource. Returns 1 on success or 0 on failure.

Parameters:

- *path.* Resource path.
- *name.* Resource name.
- *rt.* Resource type.
- *ifd.* Resource interface descriptor.
- *sz.* Resource estimated size.
- *is_subscription.* Indicates if the resource is able to handle subscriptions.
- *handler.* Handler to return a response given a request.
- *resource.*Return pointer for the pointer to the allocated resource.

```
int coap_resource_add_handler(coap_resource_t *resource,
        void (*handler)(coap_pdu_t *));
```

Sets a new handler to a resource. Returns 1 on success or 0 on failure.

Parameters:

- *handler.* Handler to return a response given a request.
- *resource.* Return pointer for the pointer to the allocated resource.

```
int coap_resource_delete(coap_resource_t *resource);
```

Removes a resource. Returns 1 on success or 0 on failure.

Parameters:

- *resource*. The resource.

```
void coap_resource_print(coap_resource_t *resource);
```

Prints a resource using the standard error.

Parameters:

- *resource*. The resource.

```
int coap_resource_link_format(coap_resource_t *resource, char *data, int *len);
```

Gives a CoRE link format representation of a resource. Returns 1 on success or 0 on failure.

Parameters:

- *resource*. The resource.
- *data*. String containing the CoRE Link format representation.
- *len*. Length of the CoRE Link format representation.

```
int coap_resource_update(coap_list_t *resources, char *data, int len);
```

Updates a list of resources given a CoRE Link format string. Returns 1 on success or 0 on failure.

Parameters:

- *resources*. The list of resources.
- *data*. String containing the CoRE Link format representation.
- *len*. Length of the CoRE Link format representation.

```
void coap_resource_node_delete(coap_node_t *node);
```

Function callback called when a resource is removed from a list.

Parameters:

- *node.* The list node.

D CoAP proxy API

D.1 connection.h

D.1.1 Data structures

```
typedef struct {
        int type;
        char *path;
        coap_pdu_t *coap_request;
        time_t timestamp;
        void *data;
} connection_t;


#define FASTCGI_UNIX_REQUEST(connection) ((FCGX_Request *)((connection)->data))
#define WEBSOCKET_SOCKET(connection) (*((int *)((connection)->data)))
#define COAP_ORIGIN_ADDR(connection) ((struct sockaddr_in6 *)((connection)->data))
#define COAP_SOCKET(connection) (*((int *)((connection)->data+sizeof(struct sockaddr_in6))))
```

D.1.2 Functions

```
int connection_create(int type, connection_t **connection);
```

Creates a new connection. Returns 1 on success or 0 on failure.

Parameters:

- *type.* Connection type (FASTCGI or WEBSOCKET).
- *connection.* Return pointer for the pointer to the allocated connection.

```
void connection_node_delete(struct coap_node_t *node);
```

Function callback to remove a connection from a list.

Parameters:

- *node*. List node to remove.

```
int connection_delete(connection_t *connection);
```

Frees a connection. Returns 1 on success or 0 on failure.

Parameters:

- *connection*. The connection to be freed.

```
void connection_print(connection_t *connection);
```

Prints a connection using the standard error output.

Parameters:

- *connection*. The connection.

```
int connection_request_send(connection_t *connection, coap_resource_t *resource);
```

Sends the CoAP PDU from a connection to a resource. Returns 1 on success or 0 on failure.

Parameters:

- *connection*. The connection that contains the CoAP PDU.
- *resource*. The resource where to send the CoAP PDU.

```
int connection_response_send(connection_t *connection, coap_pdu_t *response);
```

Sends a response back to the requester. The requester can be connected through the FastCGI or WebSocket interface. Returns 1 on success or 0 on failure.

Parameters:

- *connection*. The connection.
- *response*. The response to be sent.

D.2 resource.h

D.2.1 Data structures

```
typedef struct {
        char *path;
        coap_resource_t *resource;
        coap_list_t *connections;
        coap_client_t *client;
} resource_t;
```

D.2.2 Functions

```
int resource_create(char *name, coap_resource_t *resource, resource_t **res);
```

Creates a new proxy resource based on a server resource. Returns 1 on success or 0 on failure.

Parameters:

- *name*. Resource name.
- *resource*. The server resource to be used in the proxy resource.
- *res*. Return pointer for the pointer to the allocated proxy resource.

```
void resource_delete(resource_t *resource);
```

Frees a proxy resource.

Parameters:

- *resource.* The proxy resource to be freed.

```
void resource_node_delete(struct coap_node_t *node);
```

Function callback to remove a proxy resource from a list.

Parameters:

- *node.* The list node.

D.3 sockset.h

D.3.1 Data structures

```
typedef struct {
        fd_set rdfs;
        struct timeval tv;
        int max;
} sockset_t;
```

D.3.2 Functions

```
void sockset_init(sockset_t *ss);
```

Initializes a socket.

Parameters:

- *ss.* The sockset to initialize.

```
void sockset_add(sockset_t *ss,int sd);
```

Adds a new socket to the sockset.

Parameters:

- *ss*. The sockset.
- *sd*. The socket to be added.

```
void sockset_remove(sockset_t *ss,int sd);
```

D.4 websocket.h

D.4.1 Functions

```
int websocket_do_handshake(int sock);
```

Performs the handshake. Returns 1 on success or 0 on failure.

Parameters:

- *sock*. The socket number to use.

```
int websocket_send(int sock, char *message);
```

Sends a message through a WebSocket connection.

Parameters:

- *sock*. The socket number to use.
- *message*. The message to be sent.

E SQLite database

```
CREATE TABLE resource_directory (
    id INTEGER PRIMARY KEY,
    name VARCHAR(60)
);
```

This table represents a resource directory identified by a *name*.

```
CREATE TABLE resource (
    id INTEGER PRIMARY KEY,
    name VARCHAR(60),
    host VARCHAR(60),
    port INTEGER,
    rt VARCHAR(65),
    if VARCHAR(65),
    resource_directory_id INTEGER,
    sz INTEGER NOT NULL DEFAULT 0,
    path TEXT,
    subscription INTEGER NOT NULL DEFAULT 0
);
```

This table models a CoAP resource. It is identified by a *name*. Each resource is associated with a host (*host*) and a port (*port*) to identify the destination server. Also the path in the destination server (*path*) and resource semantic parameters described in 3.3.1 (*rt*, *if*, *sz*) are stored in this table.

```
CREATE TABLE resource_mapping (
    id INTEGER PRIMARY KEY,
    resource_id INTEGER,
    path VARCHAR(60)
);
```

This table maps an existing resource with an URL that will be used by the Web browser to access the CoAP network resource.