ΟΙΚΟΝΟΜΙΚΟ ΠΑΝΕΠΙΣΤΗΜΙΟ ΑΘΗΝΩΝ

ATHENS UNIVERSITY OF ECONOMICS AND BUSINESS

ΤΜΗΜΑ ΠΛΗΡΟΦΟΡΙΚΗΣ

ΜΕΤΑΠΤΥΧΙΑΚΟ ΔΙΠΛΩΜΑ ΕΙΔΙΚΕΥΣΗΣ (MSc)

στα ΠΛΗΡΟΦΟΡΙΑΚΑ ΣΥΣΤΗΜΑΤΑ

ΔΙΠΛΩΜΑΤΙΚΗ ΕΡΓΑΣΙΑ

“Experiments with COAP over L2 technologies”

Ηλίας Καρδίτσης

MM414004

ΑΘΗΝΑ, ΙΟΥΛΙΟΣ 2016

Abstract

In a world where the Internet of Things (IoT) becomes more and more popular, there is always a need for more targeted ways for constrained devices to communicate. The Constrained Application Protocol (COAP) was designed for that purpose, targeting small, low-power sensors, switches, valves and similar components that need to be controlled or supervised remotely, through standard Internet networks. It was designed by The Internet Engineering Task Force (IETF) Constrained RESTful environments (CoRE) Working Group with the following features in mind: Overhead and parsing complexity, URI and content-type support, Support for the discovery of resources provided by known CoAP services, Simple subscription for a resource, and resulting push notifications and Simple caching based on max-age. Although COAP can in principle be implemented over any type of network technology ensuring end-to-end connectivity, it is usually implemented on top of UDP, which means it also requires IP to be implemented underneath. In this thesis, we have implemented an experiment where a COAP packet is created and injected directly over a Layer 2 technology (Ethernet). The packet is received by a Raspberry Pi in order to be decoded and for some predefined operations to be performed, based on the packet's payload.

TABLE OF CONTENTS

Abstract 2

Acknowledgments 5

1. **Introduction** 6

2. **IOT existing protocols** 7

- **MQTT** 7
- **COAP** 8

3. **COAP Protocol specification.** 9

4. **COAP Request for comments.** 10

A. Constrained Application Protocol 10

a. Messaging Model 10

b. Request/Response 11

c. Intermediaries and Caching 13

B. Message Format 13

a. Option Format 15

C. Message Transmission 16

5. **Ethernet frame** 17

a. Frame format 17

6. **UDP Packet** 18

a. UDP Packet Structure 18

7. **COAP –Ethernet frame encapsulation** 20

a. Libraries used 20

b. Source code 20

i. Cient 20

ii. Server 20

c. Expected Results 21

d. Outcome 22

Appendix 23

TABLE OF FIGURES

Figure 1: MQTT Architecture 7
Figure 2:MQTT Architecture 8
Figure 3:Reliable Image Transmission 10
Figure 4:Two GET Requests with Piggybacked Responses 11
Figure 5:A GET Request with a Separate Response 12
Figure 6:A Request and a Response Carried in Non-confirmable 12
Figure 7:COAP Message Format 14
Figure 8: Option Format 15
Figure 9:Ethernet Frame 17
Figure 10:UDP Packet 19

Acknowledgments

I wish to thank my supervisor for providing his valuable guidance and advice throughout this project. I would also like to thank my family for both their moral and financial support during the years of my studies. Finally, I would like to dedicate this paper to the memory of my father.

1. Introduction

The Internet of Things (IoT) is a system of interrelated computing devices, mechanical and digital machines, objects, animals or people that are provided with unique identifiers and the ability to transfer data over a network without requiring human-to-human or human-to-computer interaction.

A thing, in the Internet of Things, can be a person with a heart monitor implant, a farm animal with a biochip transponder, an automobile that has built-in sensors to alert the driver when tire pressure is low, or any other natural or man-made object that can be assigned an IP address and provided with the ability to transfer data over a network.

IoT has evolved from the convergence of wireless technologies, micro-electromechanical systems (MEMS), microservices and the Internet. The convergence has helped tear down the silo walls between operational technology (OT) and information technology (IT), allowing unstructured machine-generated data to be analyzed for insights that will drive improvements.

2. IOT existing protocols

The IoT needs standard protocols. Two of the most promising for small devices are MQTT and CoAP. Both MQTT and CoAP:

- Are open standards
- Are better suited to constrained environments than HTTP
- Provide mechanisms for asynchronous communication
- Run on top of IP
- Have a range of implementations

MQTT offers flexibility in communication patterns and acts purely as a pipe for binary data. On the other hand, CoAP is designed for interoperability with the web.

- **MQTT**

MQTT is a publish/subscribe messaging protocol designed for lightweight M2M communications. It was originally developed by IBM and is now an open standard.

MQTT offers a client/server model, where every sensor is a client and connects to a server, known as a broker, over TCP. MQTT is message oriented. Every message is a discrete chunk of data, opaque to the broker. Every message is published to an address, known as a topic. Clients may subscribe to multiple topics. Every client subscribed to a topic, receives every message published to the topic. For example, imagine a simple network with three clients and a central broker. All three clients open TCP connections with the broker. Clients B and C subscribe to the topic temperature.

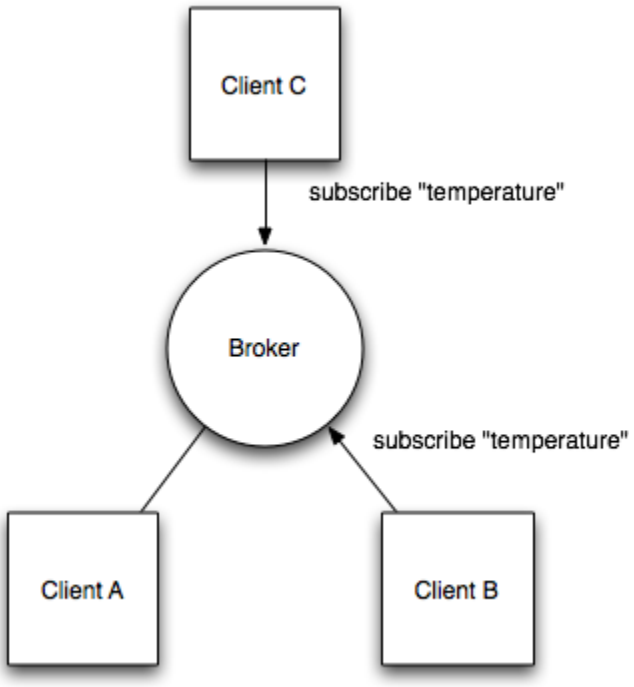

Figure 1: MQTT Architecture

At a later time, Client A publishes a value of 22.5 for the topic temperature. The broker forwards the message to all subscribed clients.

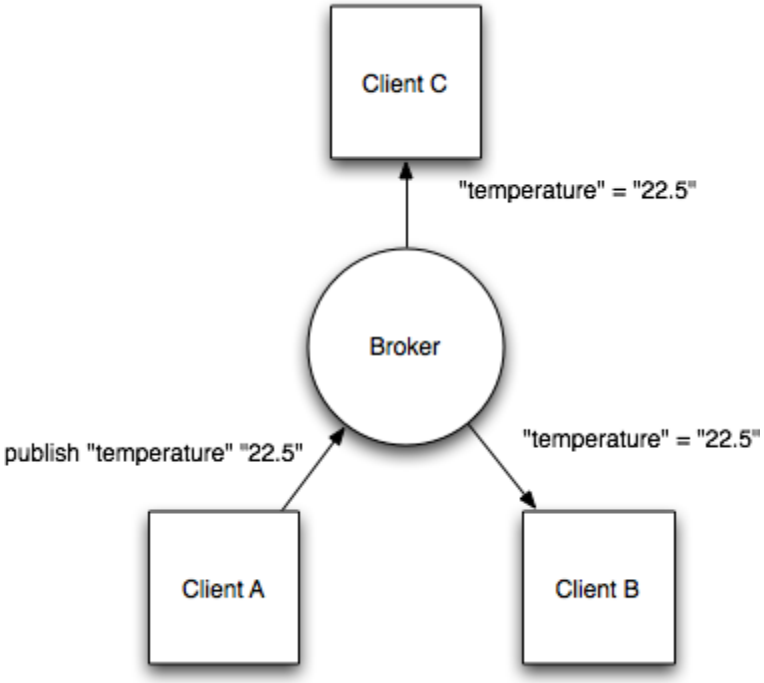

Figure 2:MQTT Architecture

The publish-subscribe model allows MQTT clients to communicate one-to-one, one-to-many and many-to-one.

- **COAP**

CoAP is the Constrained Application Protocol from the CoRE (Constrained Resource Environments) IETF group.

Like HTTP, CoAP is a document transfer protocol. Unlike HTTP, CoAP is designed for the needs of constrained devices. CoAP packets are much smaller than HTTP TCP flows. Bitfields and mappings from strings to integers are used extensively to save space. Packets are simple to generate and can be parsed in place without consuming extra RAM in constrained devices. CoAP runs over UDP, not TCP. Clients and servers communicate through connectionless datagrams. Retries and reordering are implemented in the application stack. Removing the need for TCP may allow full IP networking in small microcontrollers. CoAP allows UDP broadcast and multicast to be used for addressing.

CoAP follows a client/server model. Clients make requests to servers, servers send back responses. Clients may GET, PUT, POST and DELETE resources. CoAP is designed to interoperate with HTTP and the RESTful web at large through simple proxies. Because CoAP is datagram based, it may be used on top of SMS and other packet based communications protocols.

3. COAP Protocol specification.

The Constrained Application Protocol (CoAP) is a specialized web transfer protocol for use with constrained nodes and constrained (e.g., low-power, lossy) networks. The nodes often have 8-bit microcontrollers with small amounts of ROM and RAM, while constrained networks such as IPv6 over Low-Power Wireless Personal Area Networks (6LoWPANs) often have high packet error rates and a typical throughput of 10s of kbit/s. The protocol is designed for machine-to-machine (M2M) applications such as smart energy and building automation.

CoAP provides a request/response interaction model between application endpoints, supports built-in discovery of services and resources, and includes key concepts of the Web such as URIs and Internet media types. CoAP is designed to easily interface with HTTP for integration with the Web while meeting specialized requirements such as multicast support, very low overhead, and simplicity for constrained environments.

4. COAP Request for comments.

A. Constrained Application Protocol

a. Messaging Model

The CoAP messaging model is based on the exchange of messages over UDP between endpoints. CoAP uses a short fixed-length binary header (4 bytes) that may be followed by compact binary options and a payload. This message format is shared by requests and responses. The CoAP message format is specified below. Each message contains a Message ID used to detect duplicates and for optional reliability. (The Message ID is compact; its 16-bit size enables up to about 250 messages per second from one endpoint to another with default protocol parameters).

Reliability is provided by marking a message as Confirmable (CON). A Confirmable message is retransmitted using a default timeout and exponential back-off between retransmissions, until the recipient sends an Acknowledgement message (ACK) with the same Message ID (in this example, 0x7d34) from the corresponding endpoint; see Figure 3. When a recipient is not at all able to process a Confirmable message (i.e., not even able to provide a suitable error response), it replies with a Reset message (RST) instead of an Acknowledgement (ACK).

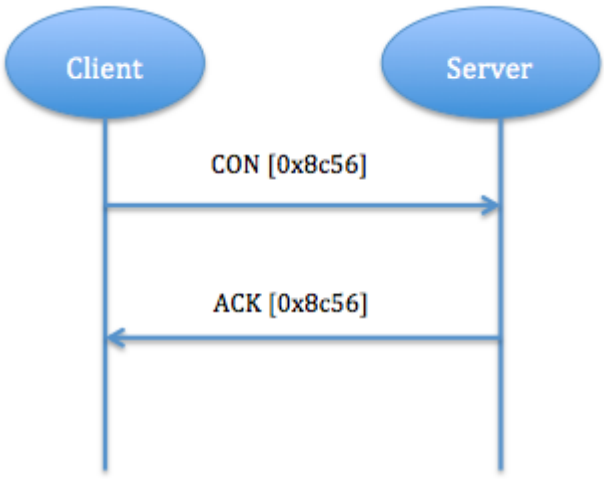

Figure 3:Reliable Image Transmission

A message that does not require reliable transmission (for example, each single measurement out of a stream of sensor data) can be sent as a Non-confirmable message (NON). These are not acknowledged, but still have a Message ID for duplicate detection. When a recipient is not able to process a Non-confirmable message, it may again reply with a Reset message (RST).

As CoAP runs over UDP, it also supports the use of multicast IP destination addresses, enabling multicast CoAP requests.

Several security modes are defined for CoAP ranging from no security to certificate-based security.

b. Request/Response

CoAP request and response semantics are carried in CoAP messages, which include either a Method Code or Response Code, respectively. Optional (or default) request and response information, such as the URI and payload media type are carried as CoAP options. A Token is used to match responses to requests independently from the underlying messages. (Note that the Token is a concept separate from the Message ID.)

A request is carried in a Confirmable (CON) or Non-confirmable (NON) message, and, if immediately available, the response to a request carried in a Confirmable message is carried in the resulting Acknowledgement (ACK) message. This is called a piggybacked response. (There is no need for separately acknowledging a piggybacked response, as the client will retransmit the request if the Acknowledgement message carrying the piggybacked response is lost.) Two examples for a basic GET request with piggybacked response are shown in Figure 4, one successful, one resulting in a 4.04 (Not Found) response.

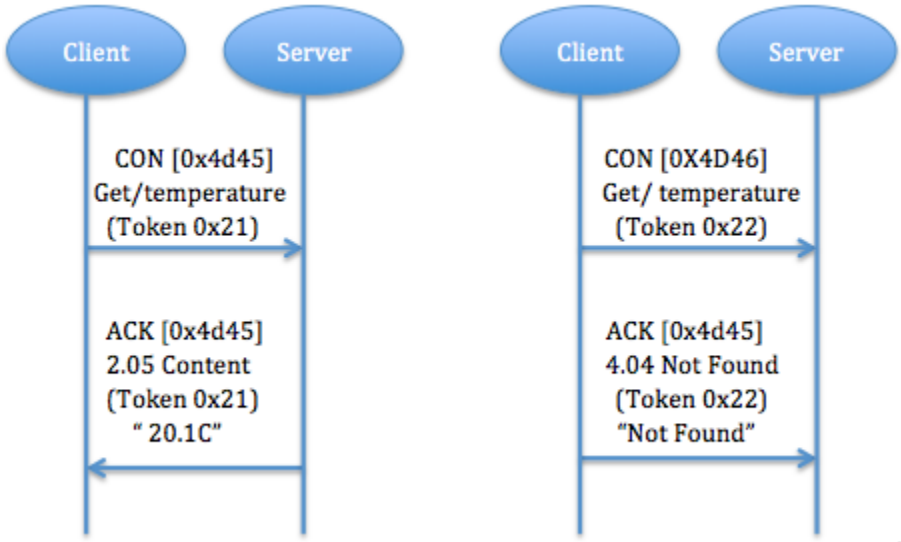

Figure 4:Two GET Requests with Piggybacked Responses

If the server is not able to respond immediately to a request carried in a Confirmable message, it simply responds with an Empty Acknowledgement message so that the client can stop retransmitting the request. When the response is ready, the server sends it in a new Confirmable message (which then in turn needs to be acknowledged by the client). This is called a "separate response", as illustrated by the following figure.

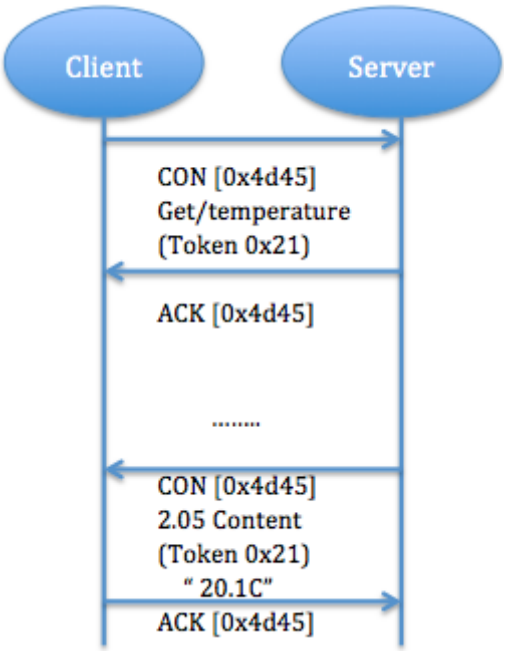

Figure 5:A GET Request with a Separate Response

If a request is sent in a Non-confirmable message, then the response is sent using a new Non-confirmable message, although the server may instead send a Confirmable message. This type of exchange is illustrated below.

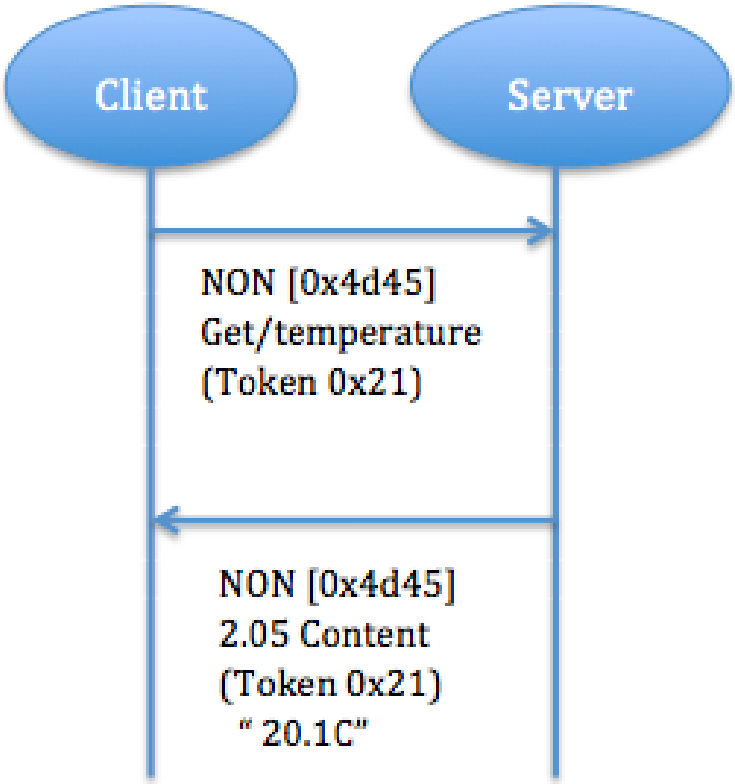

Figure 6:A Request and a Response Carried in Non-confirmable

CoAP makes use of GET, PUT, POST, and DELETE methods in a similar manner to HTTP. (Note that the detailed semantics of CoAP methods are "almost, but not entirely unlike those of HTTP methods: intuition

taken from HTTP experience generally does apply well, but there are enough differences that make it worthwhile to actually read the present specification).

Methods beyond the basic four can be added to CoAP in separate specifications. New methods do not necessarily have to use requests and responses in pairs. Even for existing methods, a single request may yield multiple responses, e.g., for a multicast request or with the Observe option.

URI support in a server is simplified as the client already parses the URI and splits it into host, port, path, and query components, making use of default values for efficiency. Response Codes relate to a small subset of HTTP status codes with a few CoAP-specific codes added.

c. Intermediaries and Caching

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

Proxying is useful in constrained networks for several reasons, including to limit network traffic, to improve performance, to access resources of sleeping devices, and for security reasons. The proxying of requests on behalf of another CoAP endpoint is supported in the protocol. When using a proxy, the URI of the resource to request is included in the request, while the destination IP address is set to the address of the proxy.

As CoAP was designed according to the REST architecture, and therefore exhibits functionality similar to that of the HTTP protocol, it is quite straightforward to map from CoAP to HTTP and from HTTP to CoAP. Such a mapping may be used to realize an HTTP REST interface using CoAP or to convert between HTTP and CoAP. This conversion can be carried out by a cross-protocol proxy ("cross-proxy"), which converts the Method or Response Code, media type, and options to the corresponding HTTP feature.

B. Message Format

CoAP is based on the exchange of compact messages that, by default, are transported over UDP (i.e., each CoAP message occupies the data section of one UDP datagram). CoAP may also be used over Datagram Transport Layer Security (DTLS). It could also be used over other transports such as SMS, TCP, or SCTP, the specification of which is out of this document's scope.

CoAP messages are encoded in a simple binary format. The message format starts with a fixed-size 4-byte header. This is followed by a variable-length Token value, which can be between 0 and 8 bytes long. Following the Token value, comes a sequence of zero or more CoAP Options in Type-Length-Value (TLV) format, optionally followed by a payload that takes up the rest of the datagram.

Table 3 Message Format

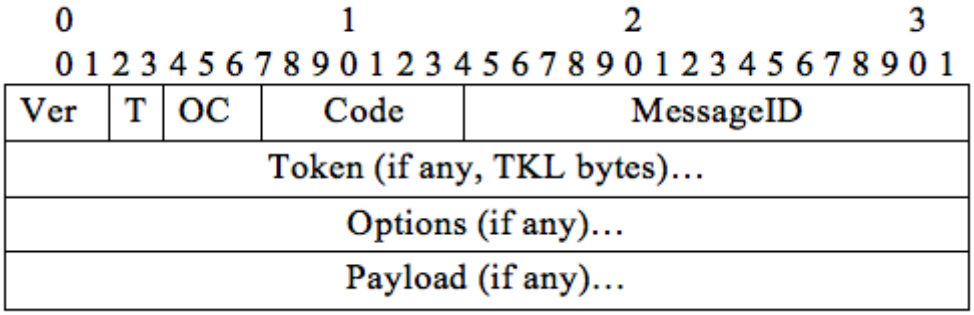

Figure 7:COAP Message Format

The fields in the header are defined as follows:

Version (Ver): 2-bit unsigned integer. It indicates the CoAP version number. Implementations of this specification MUST set this field to 1 (01 binary). Other values are reserved for future versions. Messages with unknown version numbers MUST be silently ignored.

Type (T): 2-bit unsigned integer. It indicates if this message is of type Confirmable (0), Non-confirmable (1), Acknowledgement (2), or Reset (3).

Token Length (TKL): 4-bit unsigned integer. It indicates the length of the variable-length Token field (0-8 bytes). Lengths 9-15 are reserved, MUST NOT be sent, and MUST be processed as a message format error.

Code: 8-bit unsigned integer, split into a 3-bit class (most significant bits) and a 5-bit detail (least significant bits), documented as "c.dd" where "c" is a digit from 0 to 7 for the 3-bit subfield and "dd" are two digits from 00 to 31 for the 5-bit subfield. The class can indicate a request (0), a success response (2), a client error response (4), or a server error response (5). (All other class values are reserved.) As a special case, Code 0.00 indicates an Empty message. In case of a request, the Code field indicates the Request Method; in case of a response, a Response Code. Possible values are maintained in the CoAP Code Registries.

Message ID: 16-bit unsigned integer in network byte order. Used to detect message duplication and to match messages of type Acknowledgement/Reset to messages of type Confirmable/Non-confirmable.

The header is followed by the Token value, which may be 0 to 8 bytes, as given by the Token Length field. The Token value is used to correlate requests and responses. Header and Token are followed by zero or more Options. An Option can be followed by the end of the message, by another Option, or by the Payload Marker and the payload.

Following the header, token, and options, if any, comes the optional payload. If present and of non-zero length, it is prefixed by a fixed, one-byte Payload Marker (0xFF), which indicates the end of options and the start of the payload. The payload data extends from after the marker to the end of the UDP datagram, i.e., the Payload Length is calculated from the datagram size. The absence of the Payload Marker denotes a zero-length payload. The presence of a marker followed by a zero-length payload MUST be processed as a message format error.

a. Option Format

CoAP defines a number of options that can be included in a message.Each option instance in a message specifies the Option Number of the defined CoAP option, the length of the Option Value, and the Option Value itself.

Instead of specifying the Option Number directly, the instances MUST appear in order of their Option Numbers and a delta encoding is used between them. The Option Number for each instance is calculated as the sum of its delta and the Option Number of the preceding instance in the message. For the first instance in a message, a preceding option instance with Option Number zero is assumed. Multiple instances of the same option can be included by using a delta of zero. Option Numbers are maintained in the "CoAP Option Numbers" registry.

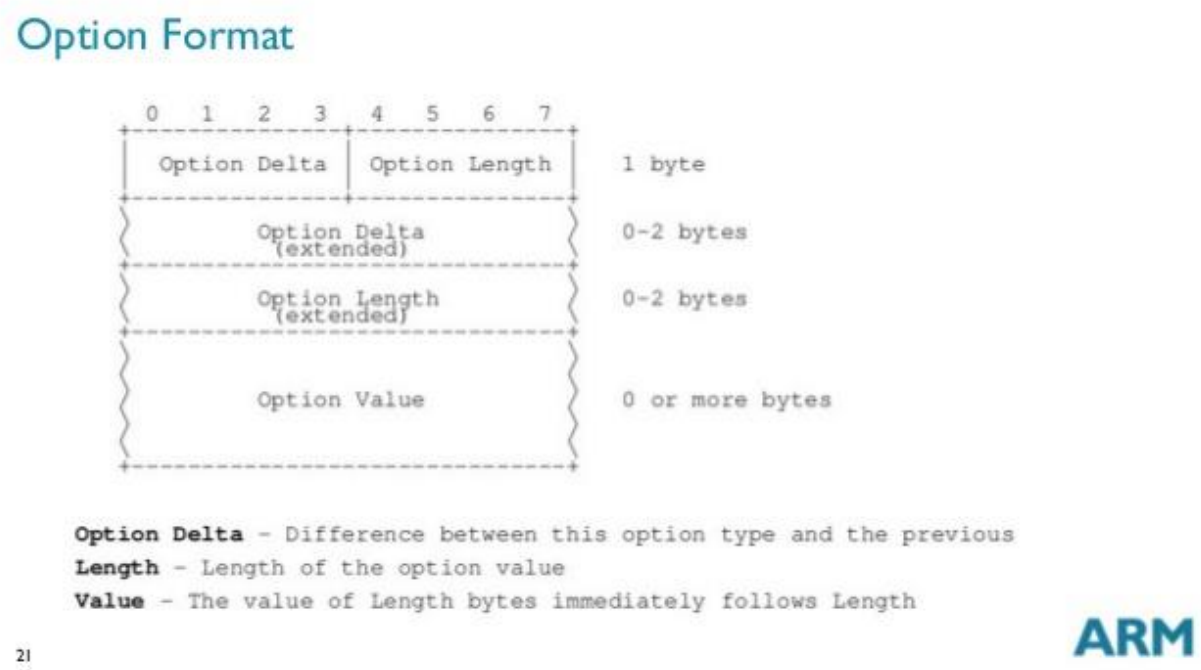

Figure 8: Option Format

The fields in an option are defined as follows:

Option Delta : 4-bit unsigned integer. A value between 0 and 12 indicates the Option Delta. Three values are reserved for special constructs:

13: An 8-bit unsigned integer follows the initial byte and indicates the Option Delta minus 13.

14: A 16-bit unsigned integer in network byte order follows the initial byte and indicates the Option Delta minus 269.

15: Reserved for the Payload Marker. If the field is set to this value but the entire byte is not the payload marker, this MUST be processed as a message format error.

The resulting Option Delta is used as the difference between the Option Number of this option and that of the previous option (or zero for the first option). In other words, the Option Number is calculated by simply summing the Option Delta values of this and all previous options before it.

Option Length: 4-bit unsigned integer. A value between 0 and 12 indicates the length of the Option Value, in bytes. Three values are reserved for special constructs:

13: An 8-bit unsigned integer precedes the Option Value and indicates the Option Length minus 13.

14: A 16-bit unsigned integer in network byte order precedes the Option Value and indicates the Option Length minus 269.

15: Reserved for future use. If the field is set to this value, it MUST be processed as a message format error.

Value: A sequence of exactly Option Length bytes. The length and format of the Option Value depend on the respective option, which MAY define variable-length values. Options defined in other documents MAY make use of other option value formats.

C. Message Transmission

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

- Simple stop-and-wait retransmission reliability with exponential back-off for Confirmable messages.
- Duplicate detection for both Confirmable and Non-confirmable messages.

5. Ethernet frame

A data packet on an Ethernet link is called an Ethernet packet, which transports an Ethernet frame as its payload. An Ethernet frame is preceded by a preamble and start frame delimiter (SFD), which are both part of the Ethernet packet at the physical layer. Each Ethernet frame starts with an Ethernet header, which contains destination and source MAC addresses as its first two fields. The middle section of the frame is payload data including any headers for other protocols (for example, Internet Protocol) carried in the frame. The frame ends with a frame check sequence (FCS), which is a 32-bit cyclic redundancy check used to detect any in-transit corruption of data.

a. Frame format

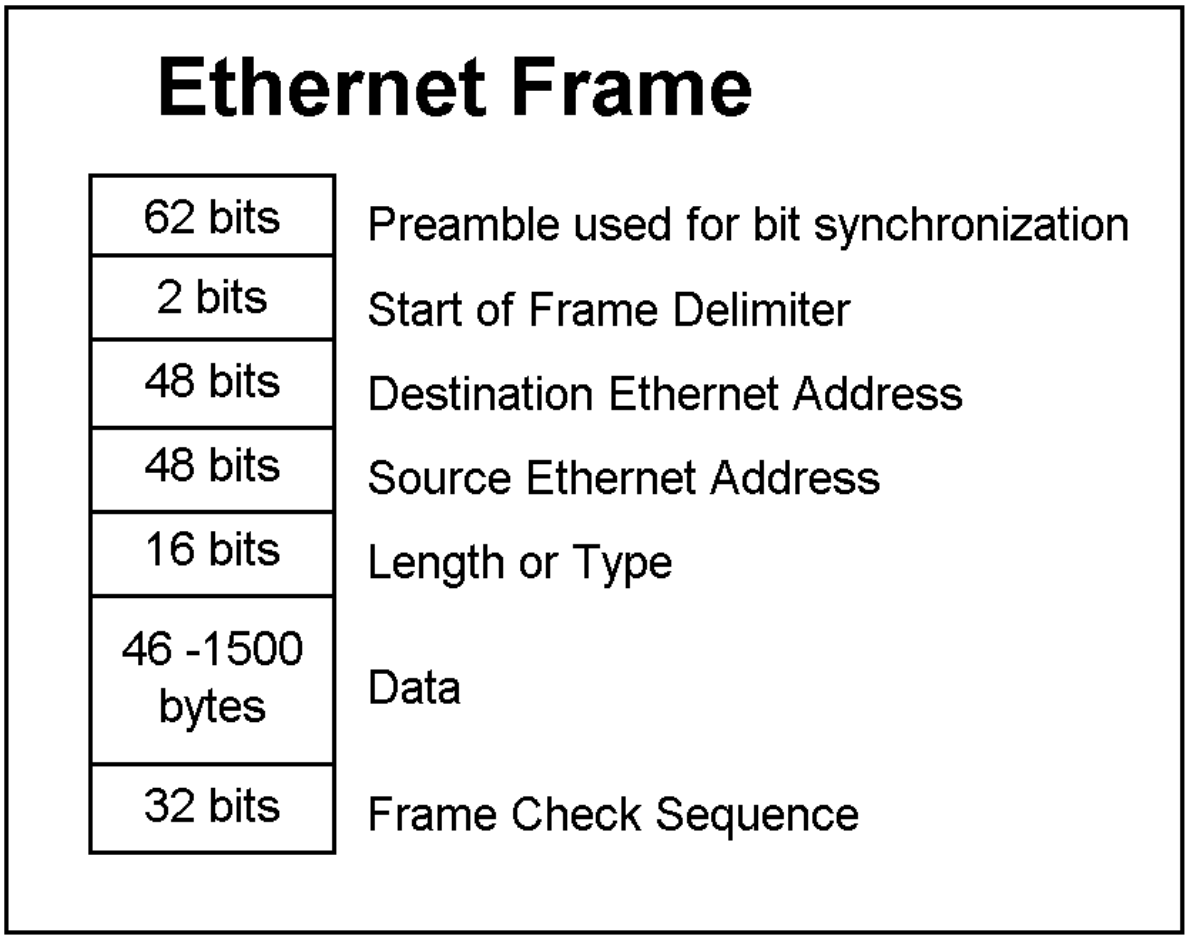

Figure 9:Ethernet Frame

6. UDP Packet

The User Datagram Protocol (UDP) is one of the core members of the Internet protocol suite. The protocol was designed by David P. Reed in 1980 and formally defined in RFC 768.

UDP uses a simple connectionless transmission model with a minimum of protocol mechanism. It has no handshaking dialogues, and thus exposes the user's program to any unreliability of the underlying network protocol. There is no guarantee of delivery, ordering, or duplicate protection. UDP provides checksums for data integrity, and port numbers for addressing different functions at the source and destination of the datagram.

With UDP, computer applications can send messages, in this case referred to as datagrams, to other hosts on an Internet Protocol (IP) network without prior communications to set up special transmission channels or data paths. UDP is suitable for purposes where error checking and correction is either not necessary or is performed in the application, avoiding the overhead of such processing at the network interface level. Time-sensitive applications often use UDP because dropping packets is preferable to waiting for delayed packets, which may not be an option in a real-time system. If error correction facilities are needed at the network interface level, an application may use the Transmission Control Protocol (TCP) or Stream Control Transmission Protocol (SCTP) which are designed for this purpose.

a. UDP Packet Structure

UDP is a minimal message-oriented Transport Layer protocol that is documented in IETF RFC 768. UDP provides no guarantees to the upper layer protocol for message delivery and the UDP layer retains no state of UDP messages once sent. Hence, it is frequently referred to as Unreliable Datagram Protocol.

UDP provides application multiplexing (via port numbers) and integrity verification (via checksum) of the header and payload. If transmission reliability is desired, it must be implemented in the user's application.

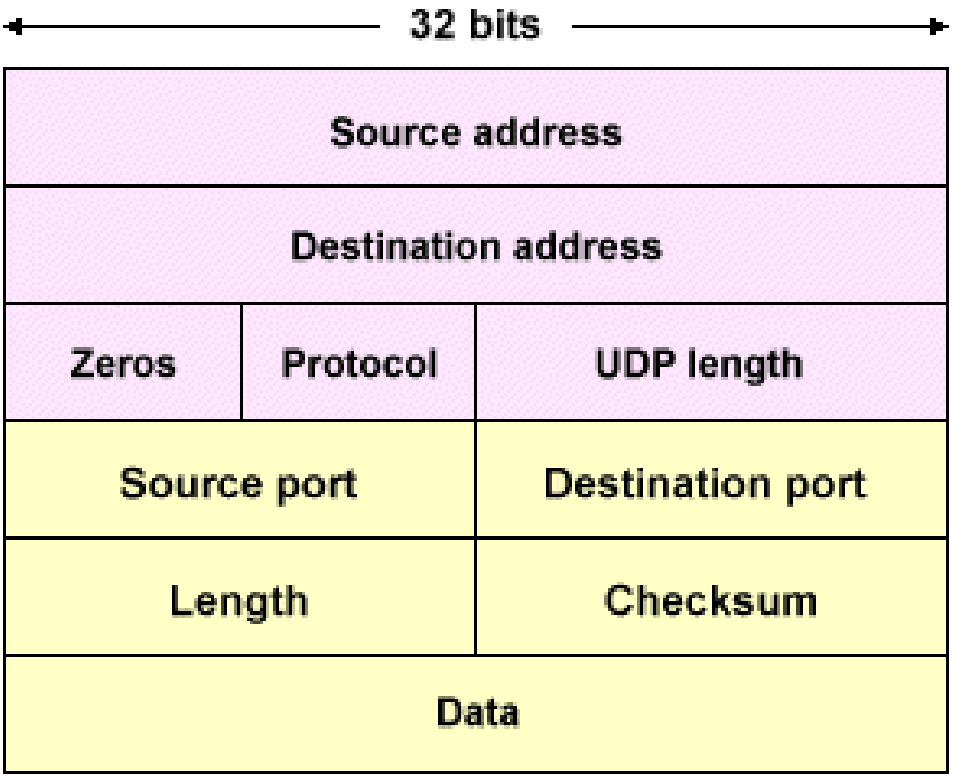

Figure 10:UDP Packet

The UDP header consists of 4 fields, each of which is 2 bytes (16 bits). The use of the fields "Checksum" and "Source port" is optional in IPv4 (pink background in table). In IPv6 only the source port is optional as it can be seen from the following.

Source port number: This field identifies the sender's port when meaningful and should be assumed to be the port to reply to if needed. If not used, then it should be zero. If the source host is the client, the port number is likely to be an ephemeral port number. If the source host is the server, the port number is likely to be a well-known port number.

Destination port number: This field identifies the receiver's port and is required. Similar to source port number, if the client is the destination host then the port number will likely be an ephemeral port number and if the destination host is the server then the port number will likely be a well-known port number.

Length: This is a field that specifies the length in bytes of the UDP header and UDP data. The minimum length is 8 bytes because that is the length of the header. The field size sets a theoretical limit of 65,535 bytes (8 byte header + 65,527 bytes of data) for a UDP datagram. The practical limit for the data length which is imposed by the underlying IPv4 protocol is 65,507 bytes (65,535 − 8 byte UDP header − 20 byte IP header).

In IPv6 jumbograms it is possible to have UDP packets of size greater than 65,535 bytes. RFC 2675 specifies that the length field is set to zero if the length of the UDP header plus UDP data is greater than 65,535.

Checksum: The checksum field may be used for error-checking of the header and data. This field is optional in IPv4, and mandatory in IPv6. The field carries all-zeros if unused.

7. COAP –Ethernet frame encapsulation

a. Libraries used

For the purposes of this experiment the following libraries were used

- ➔ Libpcap
- ➔ Libcoap.

b. Source code

i. Cient

The structs defining Ethernet, udp and coap messages are as follows :

struct ether_header *eh = (struct ether_header *) sendbuf;

struct iphdr *iph = (struct iphdr *) (sendbuf + sizeof(struct ether_header));

struct udphdr *udph = (struct udphdr *) (sendbuf + sizeof(struct iphdr) + sizeof(struct ether_header));

coap_pdu_t *pdu = (coap_pdu_t *) (sendbuf + sizeof(struct udphdr)+sizeof(struct iphdr) + sizeof(struct ether_header));

ii. Server

```
int main(int argc, char* argv[]){
   coap_context_t  *ctx;
   coap_address_t serv_addr;
   coap_resource_t *index;
   fd_set readfds;

   /* Prepare the CoAP server socket */
   coap_address_init(&serv_addr);
   serv_addr.addr.sin.sin_family = AF_INET;
   serv_addr.addr.sin.sin_addr.s_addr = INADDR_ANY;
   serv_addr.addr.sin.sin_port = htons(4000); //This is the port
   ctx = coap_new_context(&serv_addr);
   if (!ctx)
      exit(EXIT_FAILURE);
   /* Initialize the index resourse */
   index = coap_resource_init(NULL, 0, 0);
   coap_register_handler(index, COAP_REQUEST_GET, index_handler);
```

```
    coap_add_resource(ctx, index);
    while (1) {
                    printf("waiting!\n");
        FD_ZERO(&readfds);
        FD_SET( ctx->sockfd, &readfds );
        /* Block until there is something to read from the socket */
        int result = select( FD_SETSIZE, &readfds, 0, 0, NULL );
        printf("packet arrived !\n");
        if ( result < 0 ) {         /* error */
            perror("select");
                            exit(EXIT_FAILURE);
        } else if ( result > 0 ) {  /* read from socket */
                            FILE *f = fopen("file.txt", "w");
if (f == NULL)
{
    printf("Error opening file!\n");
    exit(1);
}

/* print some text */
//const char *text = "Write this to the file";
fprintf(f, "Result:%d",result);



fclose(f);
            if ( FD_ISSET( ctx->sockfd, &readfds ) )
                coap_read( ctx );
        }
    }

}
```

c. Expected Results

A CoAP packet over Ethernet is sent. Accordingly, the server creates a local file recording whether a CoAP packet has been received or not.

d. Outcome

Appendix

➔ CLIENT

```
#include <netinet/in.h>
#include <sys/socket.h>
#include <arpa/inet.h>
#include <net/if.h>
#include <netinet/ip.h>
#include <netinet/udp.h>
#include <netinet/ether.h>
#include <linux/if_packet.h>
#include <arpa/inet.h>
#include <linux/if_packet.h>
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <sys/ioctl.h>
#include <sys/socket.h>
#include <net/if.h>
#include <netinet/ether.h>
#include "coap.h"
#define MY_DEST_MAC0    0x00
#define MY_DEST_MAC1    0x00
#define MY_DEST_MAC2    0x00
#define MY_DEST_MAC3    0x00
#define MY_DEST_MAC4    0x00
#define MY_DEST_MAC5    0x00

#define DEFAULT_IF          "eth0"
#define BUF_SIZ             1024
coap_block_t block = { .num = 0, .m = 0, .szx = 6 };

static unsigned char _token_data[8];
str the_token = { 0, _token_data };

typedef unsigned char method_t;
method_t method = 1;            /* the method we are using in our requests */
```

```
static str payload = { 0, NULL }; /* optional payload to send */
/*struct ether_coap {
int ver; //COAP Version
int TKL; //COAP Token lenght
int Code; //COAP Code
int Message_ID; //COAP Message ID
int Options;
int messages;
int Payload;
int token;
char T[] ; //COAP Type
};*/


unsigned char msgtype = COAP_MESSAGE_CON; /* usually, requests are sent confirmable */

coap_pdu_t *
coap_new_request2( method_t m) ;

int main(int argc, char *argv[]){
        coap_context_t  *ctx = NULL;

        char ifName[IFNAMSIZ];
        /* Get interface name */
        if (argc > 1)
                strcpy(ifName, argv[1]);
        else
                strcpy(ifName, DEFAULT_IF);



        int sockfd;
/* Open RAW socket to send on */
if ((sockfd = socket(AF_PACKET, SOCK_RAW, IPPROTO_RAW)) == -1) {
   perror("socket");
}

//Get the index of the interface to send on:
struct ifreq if_idx;
```

```
memset(&if_idx, 0, sizeof(struct ifreq));
strncpy(if_idx.ifr_name,ifName, IFNAMSIZ-1);
if (ioctl(sockfd, SIOCGIFINDEX, &if_idx) < 0)
    perror("SIOCGIFINDEX");


//Get the MAC address of the interface to send on:
struct ifreq if_mac;

memset(&if_mac, 0, sizeof(struct ifreq));
strncpy(if_mac.ifr_name,ifName, IFNAMSIZ-1);
if (ioctl(sockfd, SIOCGIFHWADDR, &if_mac) < 0)
    perror("SIOCGIFHWADDR");

// Get the IP address of the interface to send on:
//struct ifreq if_mac;

memset(&if_mac, 0, sizeof(struct ifreq));
strncpy(if_mac.ifr_name, ifName, IFNAMSIZ-1);
if (ioctl(sockfd, SIOCGIFINDEX, &if_mac) < 0)
    perror("SIOCGIFHWADDR");

//Construct the Ethernet header:
int tx_len = 0;
char sendbuf[1024];
struct ether_header *eh = (struct ether_header *) sendbuf;

memset(sendbuf, 0, 1024);
/* Ethernet header */
eh->ether_shost[0] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[0];
eh->ether_shost[1] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[1];
eh->ether_shost[2] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[2];
eh->ether_shost[3] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[3];
eh->ether_shost[4] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[4];
eh->ether_shost[5] = ((uint8_t *)&if_mac.ifr_hwaddr.sa_data)[5];
eh->ether_dhost[0] = MY_DEST_MAC0;
eh->ether_dhost[1] = MY_DEST_MAC1;
eh->ether_dhost[2] = MY_DEST_MAC2;
eh->ether_dhost[3] = MY_DEST_MAC3;
```

```
eh->ether_dhost[4] = MY_DEST_MAC4;
eh->ether_dhost[5] = MY_DEST_MAC5;
eh->ether_type = htons(ETH_P_IP);
tx_len += sizeof(struct ether_header);

//Construct the IP header:



struct iphdr *iph = (struct iphdr *) (sendbuf + sizeof(struct ether_header));

/* IP Header */
iph->ihl = 5;
iph->version = 4;
iph->tos = 16; // Low delay
iph->id = htons(54321);
//iph->ttl = ttl; // hops
iph->ttl = 1; // hops
iph->protocol = 17; // UDP
/* Source IP address, can be spoofed */

//iph->saddr = inet_addr(inet_ntoa(((struct sockaddr_in *)&if_ip.ifr_addr)->sin_addr));
iph->saddr = inet_addr("192.168.1.7");
// iph->saddr = inet_addr("192.168.0.112");
/* Destination IP address */
iph->daddr = inet_addr("192.168.1.7");
tx_len += sizeof(struct iphdr);



//Construct the UDP header:
struct udphdr *udph = (struct udphdr *) (sendbuf + sizeof(struct iphdr) + sizeof(struct
ether_header));

/* UDP Header */
udph->source = htons(3423);
udph->dest = htons(4000);
udph->check = 0; // skip
tx_len += sizeof(struct udphdr);

//SEND COAP DATA
```

```
/*struct ether_coap *coap_req= (struct ether_coap *) (sendbuf + sizeof(struct udphdr)+sizeof(struct iphdr) + sizeof(struct ether_header));;
coap_req->ver=1;
strcpy(coap_req->T, "NON");
tx_len += sizeof(struct udphdr);*/
//sendbuf[tx_len++] = coap_req.ver;
//sendbuf[tx_len++] = coap_req.T;


 /*
  coap_hdr_t *hpdu=(coap_hdr_t *) (sendbuf +  sizeof(struct udphdr)+sizeof(struct iphdr) + sizeof(struct ether_header));

  char payload[]="O";



  hpdu->type = msgtype;
  hpdu->code = COAP_REQUEST_POST;
  hpdu->token_length = the_token.length;
  hpdu->id =  htons(1);
  tx_len += sizeof(coap_hdr_t);

  coap_pdu_t *pdu=(coap_pdu_t *) (sendbuf +  sizeof(struct udphdr)+sizeof(struct iphdr) + sizeof(struct ether_header)+ sizeof( coap_hdr_t));




  coap_add_data(pdu, sizeof(payload), payload);

  tx_len += sizeof(coap_pdu_t);
  */

  coap_pdu_t *pdu = (coap_pdu_t *) (sendbuf +  sizeof(struct udphdr)+sizeof(struct iphdr) + sizeof(struct ether_header));
  pdu = coap_new_request2( method);
```

```
  tx_len += sizeof(coap_pdu_t);
/*int TKL; //COAP Token lenght
int Code; //COAP Code
int Message_ID; //COAP Message ID
int Options;
int messages;
int Payload;
int token;*/


//tx_len += sizeof(struct ether_coap);
/* Packet data */
//sendbuf[tx_len++] = RawToBytes(&coap_req, sizeof(coap_req));
//sendbuf[tx_len+sizeof(coap_req)] =
/*sendbuf[tx_len++] = 0xad;
sendbuf[tx_len++] = 0xbe;
sendbuf[tx_len++] = 0xef;*/

unsigned short csum(unsigned short *buf, int nwords)
{
   unsigned long sum;
   for(sum=0; nwords>0; nwords--)
      sum += *buf++;
   sum = (sum >> 16) + (sum &0xffff);
   sum += (sum >> 16);
   return (unsigned short)(~sum);
}

/* Length of UDP payload and header */
udph->len = htons(tx_len - sizeof(struct ether_header) - sizeof(struct iphdr));
/* Length of IP payload and header */
iph->tot_len = htons(tx_len - sizeof(struct ether_header));
/* Calculate IP checksum on completed header */
iph->check = csum((unsigned short *)(sendbuf+sizeof(struct ether_header)), sizeof(struct
iphdr)/2);
```

```
/* Destination address */
struct sockaddr_ll socket_address;

/* Index of the network device */
socket_address.sll_ifindex = if_idx.ifr_ifindex;
/* Address length*/
socket_address.sll_halen = ETH_ALEN;
/* Destination MAC */
socket_address.sll_addr[0] = MY_DEST_MAC0;
socket_address.sll_addr[1] = MY_DEST_MAC1;
socket_address.sll_addr[2] = MY_DEST_MAC2;
socket_address.sll_addr[3] = MY_DEST_MAC3;
socket_address.sll_addr[4] = MY_DEST_MAC4;
socket_address.sll_addr[5] = MY_DEST_MAC5;
/* Send packet */
if (sendto(sockfd, sendbuf, tx_len, 0, (struct sockaddr*)&socket_address, sizeof(struct sockaddr_ll)) < 0)
   printf("Send failed\n");

}



coap_pdu_t *
coap_new_request2( method_t m) {
  coap_pdu_t *pdu;


  if ( ! ( pdu = coap_new_pdu() ) )
    return NULL;

  pdu->hdr->type = msgtype;
  pdu->hdr->id = htons(1);
  pdu->hdr->code = m;

  pdu->hdr->token_length = the_token.length;
  if ( !coap_add_token(pdu, the_token.length, the_token.s)) {
```

```
    debug("cannot add token to request\n");
  }

  coap_show_pdu(pdu);

 /* for (opt = options; opt; opt = opt->next) {
    coap_add_option(pdu, COAP_OPTION_KEY(*(coap_option *)opt->data),
                         COAP_OPTION_LENGTH(*(coap_option *)opt->data),
                         COAP_OPTION_DATA(*(coap_option *)opt->data));
  }*/

/*  if (payloadpayload.length) {
    if ((flags & FLAGS_BLOCK) == 0)
      coap_add_data(pdu, payload.length, payload.s);
    else
      coap_add_block(pdu, payload.length, payload.s, block.num, block.szx);
  }*/
   coap_add_data(pdu, payload.length, payload.s);
 //coap_add_block(pdu, payload.length, payload.s, block.num, block.szx);
  return pdu;
}
```

➔ SERVER

```
#include <netdb.h>
#include <stdio.h>
#include "coap.h"

/**
 * This function prepares the index resource
 */
static void
index_handler(coap_context_t *ctx,
              struct coap_resource_t *resource,
              const coap_endpoint_t *local_interface,
              coap_address_t *peer,
              coap_pdu_t *request,
              str *token,
              coap_pdu_t *response) {
```

```
    const char* index = "Hello World!";
    unsigned char buf[3];
    response->hdr->code = COAP_RESPONSE_CODE(205);
    coap_add_option(response,
                COAP_OPTION_CONTENT_TYPE,
                coap_encode_var_bytes(buf, COAP_MEDIATYPE_TEXT_PLAIN), buf);
    coap_add_option(response,
                COAP_OPTION_MAXAGE,
                coap_encode_var_bytes(buf, 0x2ffff), buf);
    coap_add_data(response, strlen(index), (unsigned char *)index);




}

int main(int argc, char* argv[]){
    coap_context_t  *ctx;
    coap_address_t serv_addr;
    coap_resource_t *index;
    fd_set readfds;

    /* Prepare the CoAP server socket */
    coap_address_init(&serv_addr);
    serv_addr.addr.sin.sin_family = AF_INET;
    serv_addr.addr.sin.sin_addr.s_addr = INADDR_ANY;
    serv_addr.addr.sin.sin_port = htons(4000); //This is the port
    ctx = coap_new_context(&serv_addr);
    if (!ctx)
        exit(EXIT_FAILURE);
    /* Initialize the index resourse */
    index = coap_resource_init(NULL, 0, 0);
    coap_register_handler(index, COAP_REQUEST_GET, index_handler);
    coap_add_resource(ctx, index);
    while (1) {
                    printf("waiting!\n");
        FD_ZERO(&readfds);
        FD_SET( ctx->sockfd, &readfds );
```

```
        /* Block until there is something to read from the socket */
        int result = select( FD_SETSIZE, &readfds, 0, 0, NULL );
        printf("packet arrived !\n");
        if ( result < 0 ) {         /* error */
            perror("select");
                                exit(EXIT_FAILURE);
        } else if ( result > 0 ) {  /* read from socket */
                                FILE *f = fopen("file.txt", "w");
if (f == NULL)
{
    printf("Error opening file!\n");
    exit(1);
}

/* print some text */
//const char *text = "Write this to the file";
fprintf(f, "Result:%d",result);



fclose(f);
            if ( FD_ISSET( ctx->sockfd, &readfds ) )
                coap_read( ctx );
        }
    }

}
```