Master's in Critical Computing Systems Engineering

LAB 2 – Constrained Application Protocol (CoAP)

Ricardo Severino
António Barros

Overview

CoAP stands for Constrained Application Protocol, is a communication protocol intended for constrained devices or constrained communication channels. It has a **client/server based model** much like HTTP, but with the bonus of enabling the **client to register as a resource observer,** unlike HTTP where the client needs to poll the server for updates. While CoAP is **built on top of UDP**, it does provide mechanisms for reliable communications.

In this LAB you will run and implement a CoAP server and client using different libraries. You will analyse the CoAP messaging and inspect the packet contents using wireshark.

As usual, at the end you will be asked to write a small report.

Overview of the CoAP

Constrained Application Protocol (**CoAP**) is a specialised Internet application protocol for constrained devices, as defined in RFC 7252. It enables those constrained devices,such as wireless sensor network nodes, to communicate with the wider Internet using similar protocols. CoAP is designed for use between devices on the same constrained network (e.g., low-power, lossy networks), between devices and general nodes on the Internet, and between devices on different constrained networks both joined by an internet. CoAP is also being used via other mechanisms, such as SMS on mobile communication networks.

CoAP is designed to easily translate to HTTP for simplified integration with the web, while also meeting specialised requirements such as multicast support, very low overhead, and simplicity. Multicast, low overhead, and simplicity are important for Internet of things (IoT) and machine-to-machine (M2M) communication, which tend to be embedded and have much less memory and power supply than traditional Internet devices have. Therefore, efficiency is very important. CoAP can run on most devices that support UDP or a UDP analogue.

The Internet Engineering Task Force (IETF) Constrained RESTful Environments Working Group (CoRE) has done the major standardization work for this protocol. In order to make the protocol suitable to IoT and M2M applications, various new functions have been added.

The use of **web services** (web APIs) on the Internet has become ubiquitous in most applications and depends on the fundamental **Representational State Transfer [REST] architecture** of the Web.

The work on Constrained RESTful Environments (CoRE) aims at realizing the REST architecture in a suitable form for the most constrained nodes (e.g., 8-bit microcontrollers with limited RAM and ROM) and networks (e.g., 6LoWPAN, [RFC4944]). Constrained networks such as 6LoWPAN support the fragmentation of IPv6 packets into small link-layer frames; however, this causes significant reduction in packet delivery probability. One design goal of CoAP has been to keep message overhead small, thus limiting the need for fragmentation.

The goal of CoAP is not to blindly compress HTTP [RFC2616], but rather to realize a subset of REST common with HTTP but optimized for M2M applications. Although CoAP could be used for refashioning simple HTTP interfaces into a more compact protocol, more importantly it also offers **features for M2M such as built-in discovery, multicast support, and asynchronous message exchanges**.

Features

CoAP has the following main features:

o Web protocol fulfilling M2M requirements in constrained environments

o UDP [RFC0768] binding with optional reliability supporting unicast and multicast requests.

o Asynchronous message exchanges.

o Low header overhead and parsing complexity.

o URI and Content-type support.

o Simple proxy and caching capabilities.

o A stateless HTTP mapping, allowing proxies to be built providing access to CoAP resources via HTTP in a uniform way or for HTTP simple interfaces to be realized alternatively over CoAP.

o Security binding to Datagram Transport Layer Security (DTLS) [RFC6347].

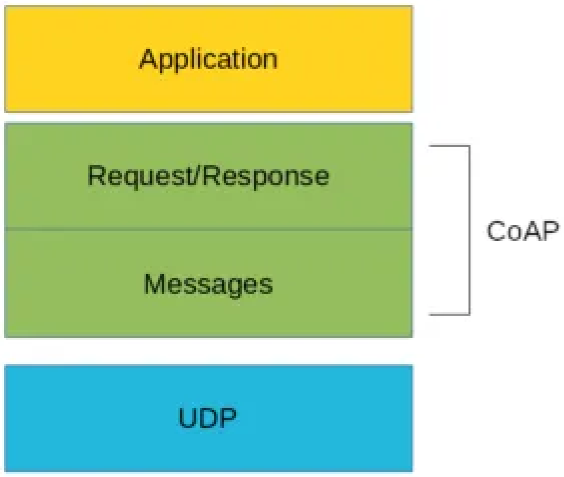
The interaction model of CoAP is similar to the client/server model of HTTP. However, machine-to-machine interactions **typically result in a CoAP implementation acting in both client and server roles**. A CoAP request is equivalent to that of HTTP and is sent by a client to request an action (using a Method Code) on a resource (identified by a URI) on a server. The server then sends a response with a Response Code; this response may include a resource representation.

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

CoAP defines four types of messages:

Confirmable, Non-confirmable, Acknowledgement, Reset.

Method Codes and Response Codes included in some of these messages make them carry requests or responses. The basic exchanges of the four types of messages are somewhat orthogonal to the request/response interactions; **requests can be carried in Confirmable and Non-confirmable messages, and responses can be carried in these as well as piggybacked in Acknowledgement messages.**

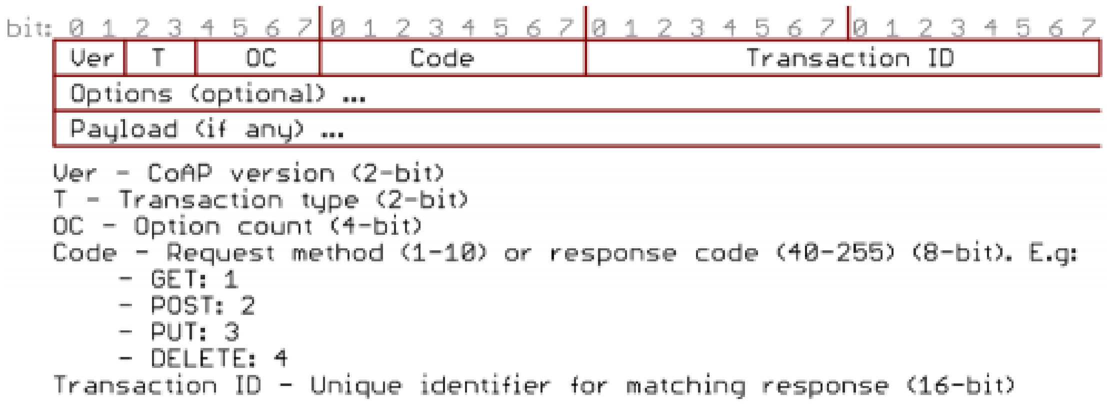


One could think of CoAP logically as using a two-layer approach, a CoAP messaging layer used to deal with UDP and the asynchronous nature of the interactions, and the request/response interactions using Method and Response Codes. CoAP is however a single protocol, with messaging and request/response as just features of the CoAP header.

Messaging Model

To avoid fragmentation, a message occupies the data section of a UDP datagram. A message is made by several parts. Each CoAP message has a unique ID; this is useful to detect message duplicates. A CoAP message is built by these parts:

- A binary header
- A compact options
- Payload

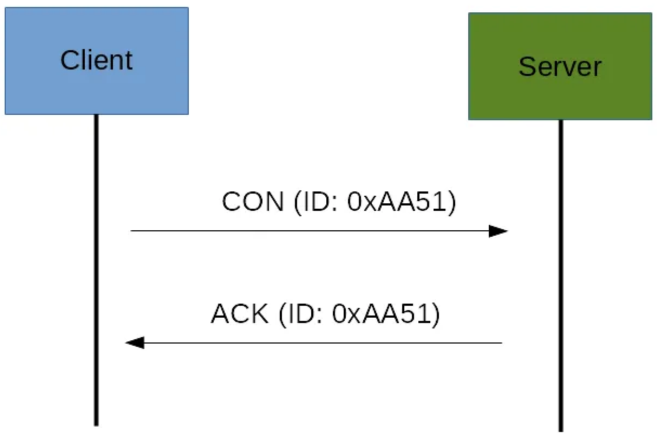

bit: 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7 0 1 2 3 4 5 6 7
Ver
T
OC
Code
Transaction ID
Options (optional) ...
Payload (if any) ...

```
Ver - CoAP version (2-bit)
T - Transaction type (2-bit)
OC - Option count (4-bit)
Code - Request method (1-10) or response code (40-255) (8-bit). E.g:
     - GET: 1
     - POST: 2
     - PUT: 3
     - DELETE: 4
Transaction ID - Unique identifier for matching response (16-bit)
```

As said before, the CoAP protocol uses two kinds of messages: Confirmable message and Non-confirmable message.

A confirmable message is a reliable message. When exchanging messages between two endpoints, these messages can be reliable. In CoAP, a reliable message is obtained using a Confirmable message (CON). Using this kind of message, the client can be sure that the message will arrive at the server. A Confirmable message is sent again and again until the other party sends an acknowledge message (ACK). The ACK message contains the same ID of the confirmable message (CON).

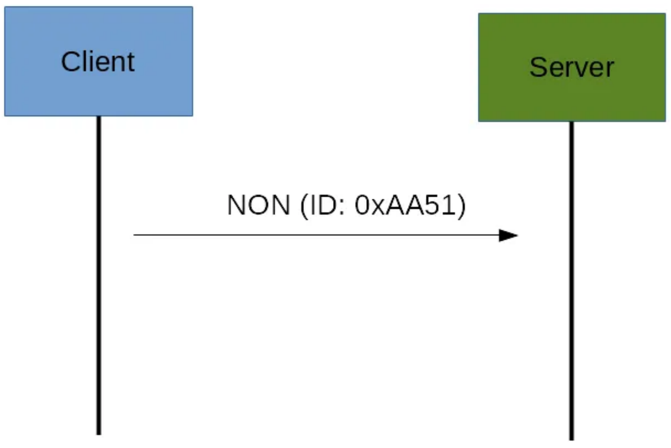
The picture below shows the message exchange process:



If the server has troubles managing the incoming request, it can send back a Reset message (RST) instead of the Acknowledge message (ACK):

The other message category is the Non-confirmable (NON) messages. These are messages that don't require an Acknowledge by the server. They are unreliable messages or in other words messages that do not contain critical information that must be delivered to the server. To this category belongs messages that contain values read from sensors.

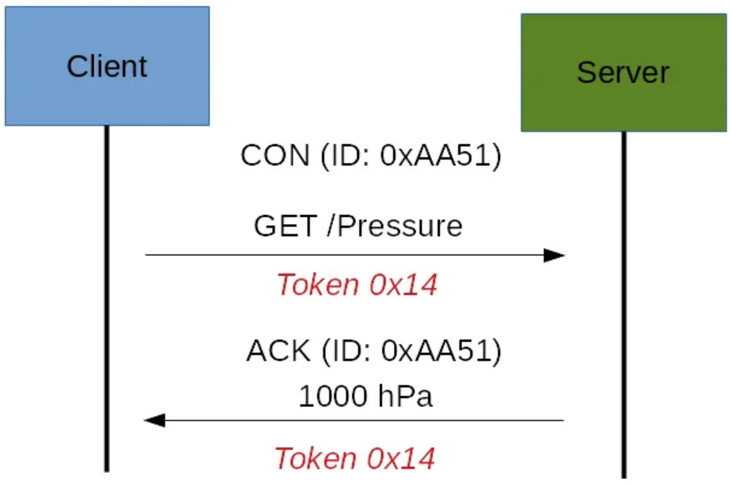
Even if these messages are unreliable, they have a unique ID.



CoAp Request/Response Model

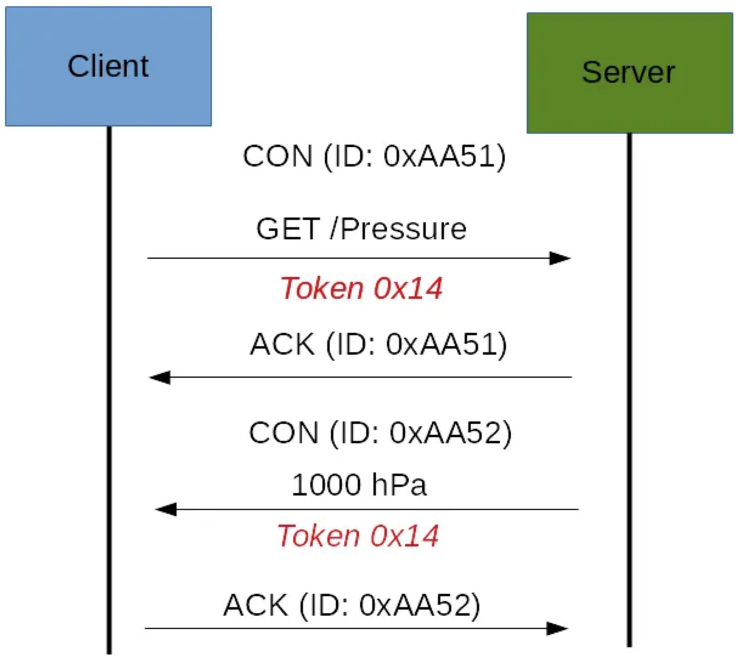
The CoAP Request/Response is the second layer in the CoAP abstraction layer. The request is sent using a Confirmable (CON) or Non-Confirmable (NON) message. There are several scenarios depending on if the server can answer immediately to the client request or the answer if not available.

If the server can answer immediately to the client request, then if the request is carried using a Confirmable message (CON), the server sends back to the client an Acknowledge message containing the response or the error code:



As you can notice in the CoAP message, there is a Token. The Token is different from the Message-ID and it is used to match the request and the response.

If the server can't answer to the request coming from the client immediately, then it sends an Acknowledge message with an empty response. As soon as the response is available, then the server sends a new Confirmable message to the client containing the response. At this point, the client sends back an Acknowledge message:



If the request coming from the client is carried using a NON-confirmable message, then the server answer using a NON-confirmable message.

CoAP Vs. MQTT

An important aspect to cover is the main differences between CoAP and MQTT. As you may know, MQTT is another protocol widely used in IoT. There are several differences between these two protocols. The first aspect to notice is the different paradigm used. MQTT uses a publisher-subscriber while CoAP uses a request-response paradigm. MQTT uses a central broker to dispatch messages coming from the publisher to the clients. CoAP is essentially a one-to-one protocol very similar to the HTTP protocol. Moreover, MQTT is an event-oriented protocol while CoAP is more suitable for state transfer.

Working with CoAP

Installing your support system

You will be installing libcoap from source. You need several dependencies in your system: **git, autoconf, pkg-config, libtool**.

In a terminal run:

```
$ git clone https://github.com/obgm/libcoap.git
$ ./autogen.sh
$ ./configure --disable-documentation --disable-dtls
$ ./configure --disable-documentation --disable-shared --without-debug CFLAGS="-D
COAP_DEBUG_FD=stderr" -- disable-dtls
$ make
$ sudo make install
```

Inspecting CoAP traffic

Open wireshark and apply the CoAP filter.

In a new terminal run you coap-server.

```
$ coap-server
```

Open a new terminal and try connecting to coap.me coap server:

```
$ coap-client -m get coap://coap.me:5683
```

Try with a confirmable and a non-confirmable message and inspect the packets you receive in wireshark.

Refer to https://libcoap.net/doc/reference/develop/man_coap-client.html to find how to do that.

Now try connecting to your local server:

```
$ coap-client -m get coap://[::1]/time
```

Explore the confirmable and non-confirmable message and check output with wireshark.

Setting you your CoAP python server

1. You need to install pip and aiocoap:

```
$ sudo apt install python3-pip
$ pip install aiocoap
```

The aiocoap package is an implementation of CoAP, the Constrained Application Protocol.

It is written in Python 3 using its native asyncio methods to facilitate concurrent operations while maintaining an easy to use interface. asyncio is a library to write concurrent code using the async/await syntax. It is used as a foundation for multiple Python asynchronous frameworks that provide high-performance network and web-servers, database connection libraries, distributed task queues, etc. It is often a perfect fit for IO-bound and high-level structured network code.

Create the server and the put client using the code below. Run and inspect in wireshark.

CoAP SERVER

```
# server.py

import aiocoap.resource as resource
import aiocoap

class AlarmResource(resource.Resource):
 """This resource supports the PUT method.
 PUT: Update state of alarm."""

 def __init__(self):
  super().__init__()
  self.state = "OFF"

 async def render_put(self, request):
   self.state = request.payload
   print('Update alarm state: %s' % self.state)

   return aiocoap.Message(code=aiocoap.CHANGED, payload=self.state)

# create a main() method to initialise the server and add the alarm resources to it.

import asyncio

async def main():
        # Resource tree creation


        root = resource.Site()
        root.add_resource(['alarm'], AlarmResource())
        await aiocoap.Context.create_server_context(root)
        await asyncio.get_running_loop().create_future()

if __name__ == "__main__":
  asyncio.run(main())
```

CoAP CLIENT

```
# client_put.py
import asyncio
import random

from aiocoap import *

async def main():
        context = await Context.create_client_context()
        alarm_state = random.choice([True, False])
        payload = b"OFF"

        if alarm_state:
                payload = b"ON"

        request = Message(code=PUT, payload=payload, uri="coap://localhost/alarm")

        response = await context.request(request).response
        print('Result: %s\n%r'%(response.code, response.payload))

if __name__ == "__main__":
  asyncio.run(main())
```

2. After testing the server and client, use the libcoap client deployed earlier to send a PUT message to your python server. Explore the libcoap options to do so.

3. Notice that the response payload is appended to the ACK message. Change the code so that the server takes more time to process the request, leading to an empty ACK response and new CON message from the server. Rely on the **await asyncio.sleep()** to do so.

Check the behaviour with wireshark.

4. Implement a new Resource in your server to process a GET request.

Going Beyond

There are several security modes defined for CoAP. Go through the relevant RFC's and other resources, and write a small report on those options. Analyse its security mechanism building on your knowledge about MQTT, from your past report.

Credits

Aiocoap Usage examples https://aiocoap.readthedocs.io/en/latest/examples.html

Creating a Simple CoAP Server with Python
https://devpress.csdn.net/python/62f4f37ec6770329307fabc6.html