Implementation and Analysis of a CoAP-Based IoT Communication System

Mohammed Alghubari - Majeed Alkhanferi - Mamdouh Alzahrani

202307030 - 202307010 - 202307270

King Fahd University

May 2025

Contents

Page

List of Figures

Page

List of Tables

1 Abstract

The Internet of Things (IoT) depends on communication protocols that are both efficient and appropriate for devices with limited resources. Conventional protocols like HTTP are too demanding on resources for IoT applications, which necessitate lighter alternatives to manage data transmission across restricted bandwidth and power. The Constrained Application Protocol (CoAP), crafted specifically for IoT, delivers a minimal overhead and a UDP-based communication method that suits low-power, low-latency settings.

This project tackles the requirement for an efficient communication system based on CoAP by implementing it on a Raspberry Pi and a Linux virtual machine. The setup comprises a CoAP server and a client developed in Python that can process GET requests and monitor resources, allowing for real-time updates. The objective was to investigate the practical application of CoAP and assess its performance in real-world scenarios.

By conducting a packet-level analysis with Wireshark, the project effectively demonstrated CoAP's low overhead and efficient communication capabilities. The findings validate that CoAP is an appropriate protocol for IoT settings, where minimal resource consumption and rapid data exchange are essential. This research underscores CoAP's efficacy as a communication protocol for IoT systems, illustrating its potential for use in constrained environments.

2 Introduction

2.1 Background Introduction

The Internet of Things (IoT) has emerged as a transformative technology, enabling smart devices to communicate over the internet. IoT applications span various domains, including healthcare, smart cities, industrial automation, and transportation. A crucial aspect of IoT is its communication infrastructure, which must support low-power, efficient, and reliable data transmission.

Traditional protocols such as HTTP and FTP are unsuitable for IoT due to their high overhead and reliance on resource-intensive transport layers. To address these challenges, lightweight protocols such as the Constrained Application Protocol (CoAP) have been introduced. CoAP, a UDP-based protocol, is optimized for constrained networks and devices, offering a RESTful interface with low overhead. It is designed to support efficient communication in resource-limited environments, making it well-suited for IoT applications.

Security remains a major concern in IoT networks, as lightweight devices are vulnerable to attacks such as unauthorized access, data interception, and denial-of-service (DoS) attacks. CoAP integrates Datagram Transport Layer Security (DTLS) to provide encryption and authentication; however, implementing security mechanisms can impact performance and introduce additional latency. Therefore, a thorough evaluation of CoAP under different network conditions is necessary to assess its effectiveness in real-world IoT deployments..

2.1.1 Problem Statement

The Internet of Things (IoT) requires efficient communication protocols for constrained devices. Traditional protocols like HTTP and FTP are too resource-intensive, making CoAP a suitable lightweight alternative. However, security risks such as unauthorized access and DoS attacks remain challenges. Additionally, DTLS-based security measures can introduce latency. Evaluating CoAP under different network conditions is essential to determine its effectiveness in real-world IoT applications.

Despite the growing use of IoT systems, real-world comparative analysis of CoAP's performance, reliability, and security is limited. Most research relies on simulations rather than actual implementations, making practical trade-offs unclear.

This project implements and evaluates a CoAP-based IoT system under various network conditions, focusing on factors like latency, energy efficiency, packet loss, and security overhead to identify its optimal use cases.

2.1.2 Aim and Objectives

Aim

The project aim is to implement and analyze a CoAP-Based IoT Communication System

Objectives

- Set up a CoAP server on a Raspberry Pi and Linux Vm.
- Implement a CoAP client to interact with the server.

- Enable resource observation for real-time data updates.
- Analyze Coap protocol.

3 Methodology

This section outlines the architecture, tools, and procedures used to implement and evaluate the CoAP-based IoT communication system.

3.0.1 Approach Overview

We deployed a CoAP server on a Raspberry Pi and implemented a Linux-based CoAP client. The client performed standard operations—GET, resource discovery, and OBSERVE. Wireshark was used to analyze protocol-level exchanges.

The evaluation compares CoAP's performance with and without DTLS under the same network environment, measuring:

- Round-trip latency
- Packet loss
- DTLS overhead

3.0.2 Tools and Environment

3.1 Tools and Environment

3.1.1 Hardware

Component	**Details**
Virtual Machine	Running Raspberry Pi OS (emulated in VM) and Linux OS

Table 3.1: Hardware components used in the test setup

3.1.2 Software

Software	**Version/Details**
Operating System	Raspberry Pi OS (emulated on a VM), Ubuntu 20.04 LTS (for the testing VM)
CoAP Library	aiocoap (Python library for CoAP)
Network Protocol	CoAP (UDP-based, for IoT communication)
Wireshark	Version 3.4.6 (used for packet capture and analysis)
Python	Python 3.x (for CoAP server and client implementation)
Twisted	Version 21.x (used for handling asynchronous network operations)
Git	Version 2.x (for managing source code and dependencies)

Table 3.2: Software components and tools used in the CoAP communication setup

3.1.3 Network Configuration

Network Type	**Details**
Communication	CoAP over UDP
CoAP Port	Default CoAP port 5683
Testing Network	Local network on the Raspberry Pi OS VM and testing VM for CoAP communication
Wireshark Interface	Monitor network traffic on localhost (127.0.0.1) and VM network

Table 3.3: Details of the network configuration used for CoAP communication

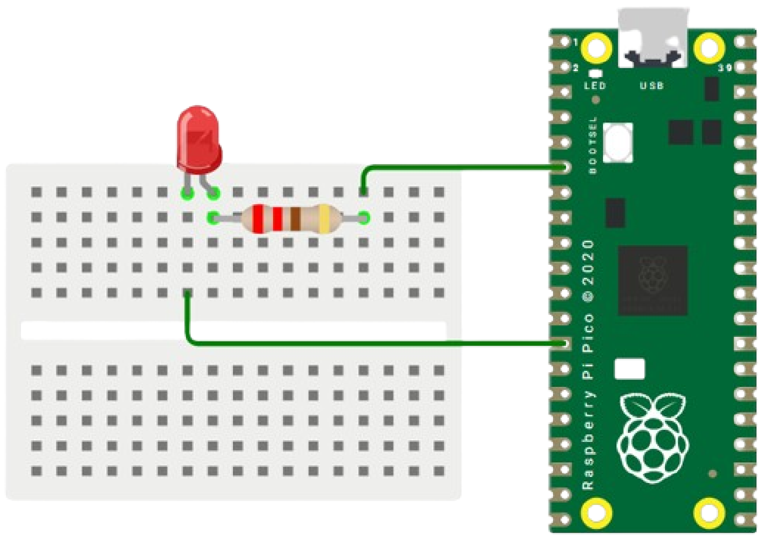


Figure 3.1: A SIMPLE RASPBERRY PI SETUP

4 Implementation

Step 1: Setting Up and Installation

- Install the latest Raspbian OS on the VM and setting up Linux Vm.
- Connect the Raspberry Pi to a network.

Installing Required Libraries On the Raspberry Pi, install `aiocoap` and `twisted`:

1. Install txThings

   ```
   Login to machine
   mkdir handsON2 && cd handsON2
   git clone git://github.com/siskin/txThings.git
   ```

2. Install Twisted Package

   ```
   pip install twisted
   ```

   If pip is not working, update it using:

   ```
   wget https://bootstrap.pypa.io/get-pip.py -O
   ./get-pip.py
   python3 ./get-pip.py
   ```

3. Install Other Dependencies

   ```
   cd txThings
   sudo python setup.py install
   ```

Step 2: Writing the CoAP Server Code

Create a Python script (`coap_server.py`) with the following code:

```
'''
Created on 30-05-2019
CoAP server runs in official IANA assigned CoAP port 5683.
@author: Bhojan Anand   (adapted from sample codes)
'''


import sys
import datetime

from twisted.internet import defer
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor
from twisted.python import log
```

```
import txthings.resource as resource
import txthings.coap as coap


class CounterResource (resource.CoAPResource):     #extends CoAPRecource class
"""
Example Resource which supports only GET method. Response is a
simple counter value.

Name render_<METHOD> is required by convention. Such method should
return a Deferred. [Python-differ module provides a small framework for asynchronous p
There are two type of callbacks: normal callbacks and errbacks, which handle an except
to use Twisted method defer.succeed(msg).
"""
#isLeaf = True

def __init__(self, start=0):
resource.CoAPResource.__init__(self)
self.counter = start
self.visible = True
self.addParam(resource.LinkParam("title", "Simple Counter Resource"))

def render_GET(self, request):
response = coap.Message(code=coap.CONTENT, payload='%d' % (self.counter,))
self.counter += 1
return defer.succeed(response)


class BlockResource (resource.CoAPResource):
"""
Example Resource which supports GET, and PUT methods. It sends large
responses, which trigger blockwise transfer (>64 bytes for normal
settings).

As before name render_<METHOD> is required by convention.
"""


def __init__(self):
resource.CoAPResource.__init__(self)
self.visible = True
self.addParam(resource.LinkParam("title", "Simple Message String Resource"))

def render_GET(self, request):
payload=" Welcome to NUS school of computing. In this course you will learn Secured In
response = coap.Message(code=coap.CONTENT, payload=payload)
return defer.succeed(response)
```

```
def render_PUT(self, request):
log.msg('PUT payload: %s', request.payload)
payload = "Just do it!."
response = coap.Message(code=coap.CHANGED, payload=payload)
return defer.succeed(response)


class SeparateLargeResource(resource.CoAPResource):
"""
Example Resource which supports GET method. It uses callLater
to force the protocol to send empty ACK first and separate response
later. Sending empty ACK happens automatically after coap.EMPTY_ACK_DELAY.
No special instructions are necessary.

Notice: txThings sends empty ACK automatically if response takes too long.

Method render_GET returns a deferred. This allows the protocol to
do other things, while the answer is prepared.

Method responseReady uses d.callback(response) to "fire" the deferred,
and send the response.
"""


def __init__(self):
resource.CoAPResource.__init__(self)
self.visible = True
self.addParam(resource.LinkParam("title", "Some Large Text Resource."))

def render_GET(self, request):
d = defer.Deferred()
reactor.callLater(3, self.responseReady, d, request)
return d

def responseReady(self, d, request):
log.msg('response ready. sending...')
payload = "Seek what you know as the highest. It does not matter whether it is going t
response = coap.Message(code=coap.CONTENT, payload=payload)
d.callback(response)

class TimeResource(resource.CoAPResource):
def __init__(self):
resource.CoAPResource.__init__(self)
self.visible = True
self.observable = True

self.notify()
self.addParam(resource.LinkParam("title", "Time now"))
```

```
def notify(self):
log.msg('TimeResource: trying to send notifications')
self.updatedState()
reactor.callLater(1, self.notify)

def render_GET(self, request):
response = coap.Message(code=coap.CONTENT, payload=datetime.datetime.now().strftime("%
return defer.succeed(response)

class CoreResource(resource.CoAPResource):
"""
Example Resource that provides list of links hosted by a server.
Normally it should be hosted at /.well-known/core

Resource should be initialized with "root" resource, which can be used
to generate the list of links.

For the response, an option "Content-Format" is set to value 40,
meaning "application/link-format". Without it most clients won't
be able to automatically interpret the link format.

Notice that self.visible is not set - that means that resource won't
be listed in the link format it hosts.
"""

def __init__(self, root):
resource.CoAPResource.__init__(self)
self.root = root

def render_GET(self, request):
data = []
self.root.generateResourceList(data, "")
payload = ",".join(data)
log.msg("%s", payload)
response = coap.Message(code=coap.CONTENT, payload=payload)
response.opt.content_format = coap.media_types_rev['application/link-format']
return defer.succeed(response)

# Resource tree creation
log.startLogging(sys.stdout)
root = resource.CoAPResource()

well_known = resource.CoAPResource()
root.putChild('.well-known', well_known)
core = CoreResource(root)
well_known.putChild('core', core)

counter = CounterResource(5000)
root.putChild('counter', counter)
time = TimeResource()
root.putChild('time', time)

other = resource.CoAPResource()
root.putChild('other', other)

block = BlockResource()
other.putChild('block', block)

separate = SeparateLargeResource()
other.putChild('separate', separate)

endpoint = resource.Endpoint(root)
reactor.listenUDP(coap.COAP_PORT, coap.Coap(endpoint)) #, interface="::")
reactor.run()
```

Step 3: Writing the CoAP Client Code
Create another Python script (`coap_client.py`) on a separate machine:

```
'''
Created on 30-05-2019
CoAP client connects to CoAP server at defauly port 5683.
@author: Bhojan Anand   (adapted from sample codes)
'''

import sys
from ipaddress import ip_address

from twisted.internet import reactor
from twisted.python import log

import txthings.coap as coap
import txthings.resource as resource


class Agent:
"""
Example class which performs single GET request to coap.me
port 5683 (official IANA assigned CoAP port), URI "test".
Request is sent 1 second after initialization.

Remote IP address is hardcoded - no DNS lookup is preformed.

Method requestResource constructs the request message to
remote endpoint. Then it sends the message using protocol.request().
A deferred 'd' is returned from this operation.

Deferred 'd' is fired internally by protocol, when complete response is received.
```

```
Method printResponse is added as a callback to the deferred 'd'. This
method's main purpose is to act upon received response (here it's simple print).
"""

def __init__(self, protocol):
self.protocol = protocol
reactor.callLater(1, self.requestResource)

def requestResource(self):
request = coap.Message(code=coap.GET)
# Send request to "coap://coap.me:5683/test"

request.opt.uri_path = ['counter']
#request.opt.uri_path = ['.well-known','core']
request.opt.observe = 0
request.remote = (ip_address("127.0.0.1"), coap.COAP_PORT)   #Change to appropriate re
d = protocol.request(request, observeCallback=self.printLaterResponse)
d.addCallback(self.printResponse)
d.addErrback(self.noResponse)

def printResponse(self, response):
print('First result: ' + response.payload)
# reactor.stop()

def printLaterResponse(self, response):
print('Observe result: ' + response.payload)

def noResponse(self, failure):
print('Failed to fetch resource:')
print(failure)
# reactor.stop()


log.startLogging(sys.stdout)

endpoint = resource.Endpoint(None)
protocol = coap.Coap(endpoint)
client = Agent(protocol)

reactor.listenUDP(61616, protocol)  # , interface="::")
reactor.run()
```

Step 4: Working with LEDs

In this step, Wokwi is use for simulation.

Requirements

- Raspberry Pi Pico board
- 1 × LED
- 1 220 ohm resistor
- 2 × female-to-female jumper wires
- 1 × mini breadboard

Python Code

Below is the Python code to blink an LED using the Raspberry Pi Pico:

```
from machine import Pin
from utime import sleep

print("Hello, Pi Pico!")

led = Pin(4, Pin.OUT)
while True:
led.toggle()
sleep(1)
```

Step 5: CoAP Service Discovery

Modify the CoAP client to discover services at the server. Change the resource list to:

```
request.opt.uri_path = ['.well-known', 'core']
```

Step 6: Coap Observable Resource

```
class CoreResource(resource.CoAPResource):
"""
Example Resource that provides list of links hosted by a server.
Normally it should be hosted at /.well-known/core

Resource should be initialized with "root" resource, which can be used
to generate the list of links.

For the response, an option "Content-Format" is set to value 40,
meaning "application/link-format". Without it most clients won't
be able to automatically interpret the link format.

Notice that self.visible is not set - that means that resource won't
be listed in the link format it hosts.
"""

def __init__(self, root):
resource.CoAPResource.__init__(self)
self.root = root

def render_GET(self, request):
data = []
self.root.generateResourceList(data, "")
payload = ",".join(data)
log.msg("%s", payload)
response = coap.Message(code=coap.CONTENT, payload=payload)
response.opt.content_format = coap.media_types_rev['application/link-format']
return defer.succeed(response)
```

Step 7: Observable Resources (Client Code)

Copy the source code from: Observable CoAP Client Code. Run the client and observe updates.

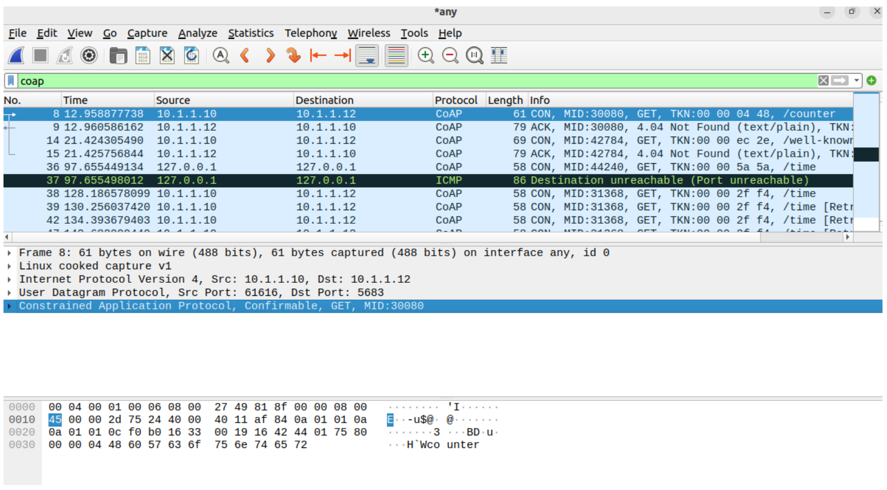
```
'''
Created on 30-05-2019
CoAP client for observable blocks
@author: Bhojan Anand   (adapted from sample codes)
'''


import sys
```

```
from twisted.internet.defer import Deferred
from twisted.internet.protocol import DatagramProtocol
from twisted.internet import reactor
from twisted.python import log

import txthings.coap as coap
import txthings.resource as resource

from ipaddress import ip_address

class Agent():
"""
Example class which performs single GET request to 'time' resource that supports obser
"""

def __init__(self, protocol):
self.protocol = protocol
reactor.callLater(1, self.requestResource)

def requestResource(self):
request = coap.Message(code=coap.GET)
#Send request to "coap://iot.eclipse.org:5683/obs-large"
request.opt.uri_path = ['time',]
request.opt.observe = 0
request.remote = (ip_address('192.168.0.115'), coap.COAP_PORT)
d = protocol.request(request, observeCallback=self.printLaterResponse,
observeCallbackArgs=('*** OBSERVE NOTIFICATION BEGIN ***',),
observeCallbackKeywords={'footer':'*** OBSERVE NOTIFICATION END ***'})
d.addCallback(self.printResponse)
d.addErrback(self.noResponse)

def printResponse(self, response):
print '*** FIRST RESPONSE BEGIN ***'
print response.payload
print '*** FIRST RESPONSE END ***'

def printLaterResponse(self, response, header, footer):
print header
print response.payload
print footer

def noResponse(self, failure):
print '*** FAILED TO FETCH RESOURCE'
print failure
#reactor.stop()

log.startLogging(sys.stdout)

endpoint = resource.Endpoint(None)
```

```
protocol = coap.Coap(endpoint)
client = Agent(protocol)

reactor.listenUDP(0, protocol)
reactor.run()
```

Step 9: Additional Practice

Running the Coap server on Rassberry Pi Vm and Coap Client in Linux VM


*any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help

```
coap

No.  Time           Source      Destination  Protocol  Length Info
   8 12.958877738   10.1.1.10   10.1.1.12    CoAP      61 CON, MID:30080, GET, TKN:00 00 04 48, /counter
   9 12.960586162   10.1.1.12   10.1.1.10    CoAP      79 ACK, MID:30080, 4.04 Not Found (text/plain), TKN:
  14 21.424305490   10.1.1.10   10.1.1.12    CoAP      69 CON, MID:42784, GET, TKN:00 00 ec 2e, /well-known
  15 21.425756844   10.1.1.12   10.1.1.10    CoAP      79 ACK, MID:42784, 4.04 Not Found (text/plain), TKN:
  36 97.655449134   127.0.0.1   127.0.0.1    CoAP      58 CON, MID:44240, GET, TKN:00 00 5a 5a, /time
  37 97.655498012   127.0.0.1   127.0.0.1    ICMP      86 Destination unreachable (Port unreachable)
  38 128.186578099  10.1.1.10   10.1.1.12    CoAP      58 CON, MID:31368, GET, TKN:00 00 2f f4, /time
  39 130.256037420  10.1.1.10   10.1.1.12    CoAP      58 CON, MID:31368, GET, TKN:00 00 2f f4, /time [Retr
  42 134.393679403  10.1.1.10   10.1.1.12    CoAP      58 CON, MID:31368, GET, TKN:00 00 2f f4, /time [Retr

Frame 8: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 4, Src: 10.1.1.10, Dst: 10.1.1.12
User Datagram Protocol, Src Port: 61616, Dst Port: 5683
Constrained Application Protocol, Confirmable, GET, MID:30080

0000  00 04 00 01 00 06 08 00  27 49 81 8f 00 00 08 00   ········ 'I······
0010  45 00 00 2d 75 24 40 00  40 11 af 84 0a 01 01 0a   E··-u$@· @·······
0020  0a 01 01 0c f0 b0 16 33  00 19 16 42 44 01 75 80   ·······3 ···BD·u·
0030  00 00 04 48 60 57 63 6f  75 6e 74 65 72            ···H`Wco unter
```

Figure 4.1: Wireshark Analysis of Coap Communication between Linux and Raspberry Pi

4.1 Adding Security To Coap

To strengthen access control and ensure confidentiality in Constrained Application Protocol (CoAP) communications, two security mechanisms have been implemented: authentication using a shared token and payload encryption using the Advanced Encryption Standard (AES).

Initially, a pre-shared token was included in the payload of each CoAP request. This token served as a static credential that allowed the server to verify the authenticity of incoming requests. However, since the token was transmitted in plaintext, it was susceptible to interception. To address this limitation, AES encryption (CBC mode) was applied to the payload, ensuring that sensitive data such as authentication tokens are protected during transmission.

This layered approach improves the resilience of the system against unauthorized access and eavesdropping, without requiring modifications to the core CoAP protocol..

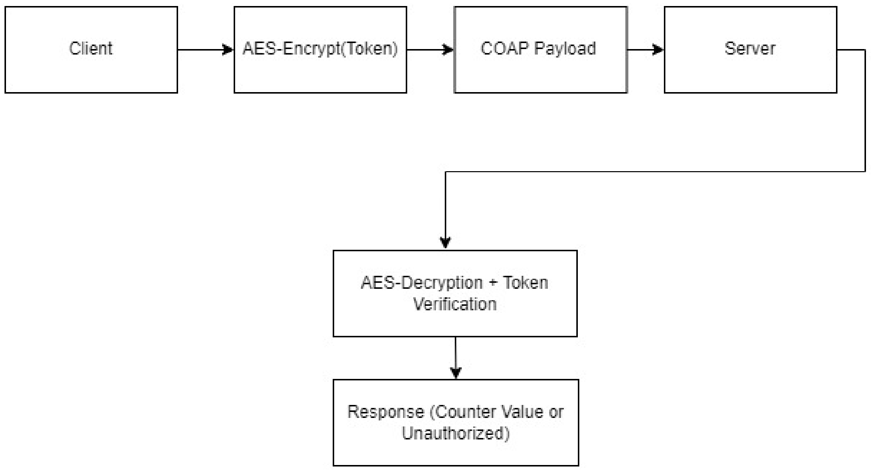

Client
AES-Encrypt(Token)
COAP Payload
Server
AES-Decryption + Token Verification
Response (Counter Value or Unauthorized)

Figure 4.2: Security Architecture

4.2 Client-Side Implementation

On the client side, the request is constructed in compliance with CoAP standards, targeting a specific resource (e.g., /counter). Prior to transmission, the authentication token is encrypted using a pre-shared AES key and Initialization Vector (IV). The encryption process includes padding the token to align with the AES block size, followed by encryption in CBC mode.

An example of client-side request construction is as follows::

```
request = coap.Message(code=coap.GET)
request.opt.uri_path = ['counter']
request.payload = encrypt_payload(SECURITY_TOKEN)
request.remote = (address, coap.COAP_PORT)
```

This ensures that the token is securely transmitted as part of the request payload

4.3 Server-Side Verification

Upon receiving a request, the CoAP server first attempts to decrypt the payload using the same AES key and IV used by the client. If decryption fails, indicating either data corruption or tampering, the server terminates the request with an UNAUTHORIZED response.

If decryption succeeds, the resulting plaintext token is compared against the expected shared secret. A successful match results in the request being processed and an appropriate response being returned. If the token is invalid, the server rejects the request.

The token verification logic is represented below:

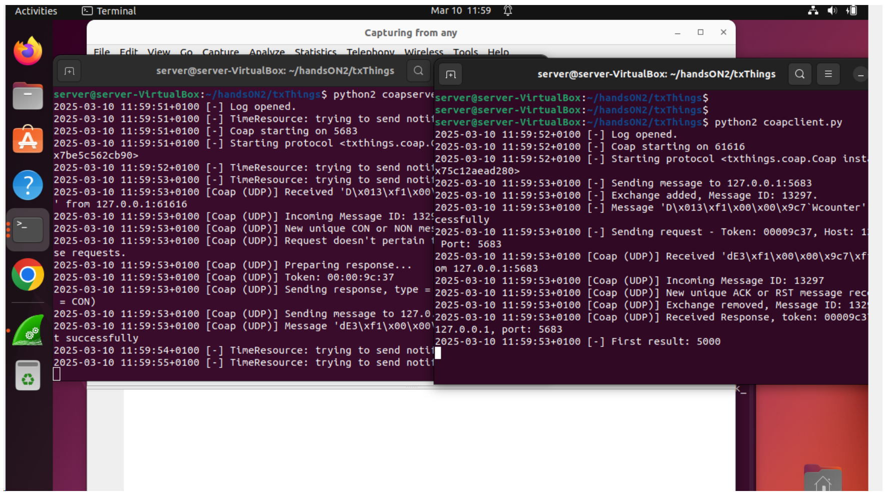
```
try:
decrypted_token = decrypt_payload(request.payload)
except:
return coap.Message(code=coap.UNAUTHORIZED,
 payload="Invalid␣token.")

if decrypted_token != SECURITY_TOKEN:
return coap.Message(code=coap.UNAUTHORIZED,
 payload="Invalid␣token.")
```

This mechanism ensures that only clients in possession of both the shared AES key and the correct token are granted access

5 Result

5.1 Capture Packet Using Wireshark

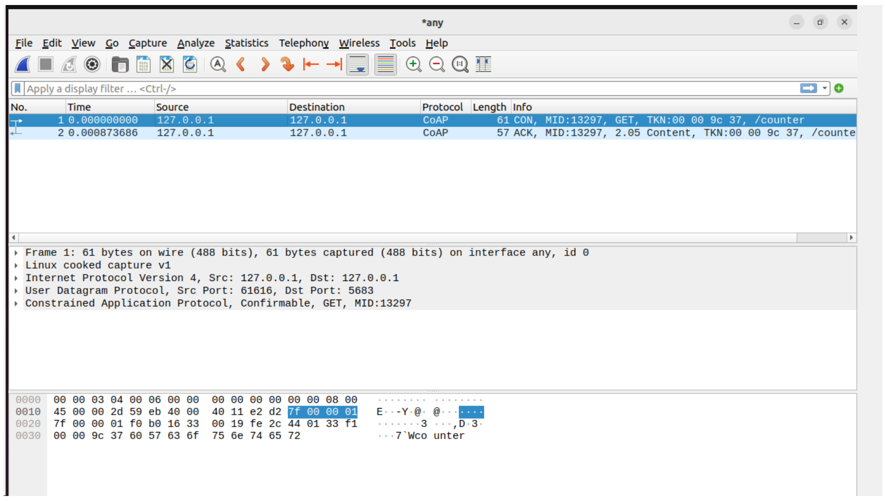
5.1.1 Running Coap Server and Client



```
server@server-VirtualBox:~/handsON2/txThings$ python2 coapserve
2025-03-10 11:59:51+0100 [-] Log opened.
2025-03-10 11:59:51+0100 [-] TimeResource: trying to send noti
2025-03-10 11:59:51+0100 [-] Coap starting on 5683
2025-03-10 11:59:51+0100 [-] Starting protocol <txthings.coap.C
x7be5c562cb90>
2025-03-10 11:59:52+0100 [-] TimeResource: trying to send noti
2025-03-10 11:59:53+0100 [-] TimeResource: trying to send noti
2025-03-10 11:59:53+0100 [Coap (UDP)] Received 'D\x013\xf1\x00\
' from 127.0.0.1:61616
2025-03-10 11:59:53+0100 [Coap (UDP)] Incoming Message ID: 132
2025-03-10 11:59:53+0100 [Coap (UDP)] New unique CON or NON mes
2025-03-10 11:59:53+0100 [Coap (UDP)] Request doesn't pertain t
se requests.
2025-03-10 11:59:53+0100 [Coap (UDP)] Preparing response...
2025-03-10 11:59:53+0100 [Coap (UDP)] Token: 00:00:9c:37
2025-03-10 11:59:53+0100 [Coap (UDP)] Sending response, type =
 = CON)
2025-03-10 11:59:53+0100 [Coap (UDP)] Sending message to 127.0.
2025-03-10 11:59:53+0100 [Coap (UDP)] Message 'dE3\xf1\x00\x00\
t successfully
2025-03-10 11:59:54+0100 [-] TimeResource: trying to send noti
2025-03-10 11:59:55+0100 [-] TimeResource: trying to send noti
```

```
server@server-VirtualBox:~/handsON2/txThings$
server@server-VirtualBox:~/handsON2/txThings$
server@server-VirtualBox:~/handsON2/txThings$ python2 coapclient.py
2025-03-10 11:59:52+0100 [-] Log opened.
2025-03-10 11:59:52+0100 [-] Coap starting on 61616
2025-03-10 11:59:52+0100 [-] Starting protocol <txthings.coap.Coap inst
x75c12aead280>
2025-03-10 11:59:53+0100 [-] Sending message to 127.0.0.1:5683
2025-03-10 11:59:53+0100 [-] Exchange added, Message ID: 13297.
2025-03-10 11:59:53+0100 [-] Message 'D\x013\xf1\x00\x00\x9c7`Wcounter'
cessfully
2025-03-10 11:59:53+0100 [-] Sending request - Token: 00009c37, Host: 1
 Port: 5683
2025-03-10 11:59:53+0100 [Coap (UDP)] Received 'dE3\xf1\x00\x00\x9c7\xf
om 127.0.0.1:5683
2025-03-10 11:59:53+0100 [Coap (UDP)] Incoming Message ID: 13297
2025-03-10 11:59:53+0100 [Coap (UDP)] New unique ACK or RST message rec
2025-03-10 11:59:53+0100 [Coap (UDP)] Exchange removed, Message ID: 132
2025-03-10 11:59:53+0100 [Coap (UDP)] Received Response, token: 00009c3
127.0.0.1, port: 5683
2025-03-10 11:59:53+0100 [-] First result: 5000
```

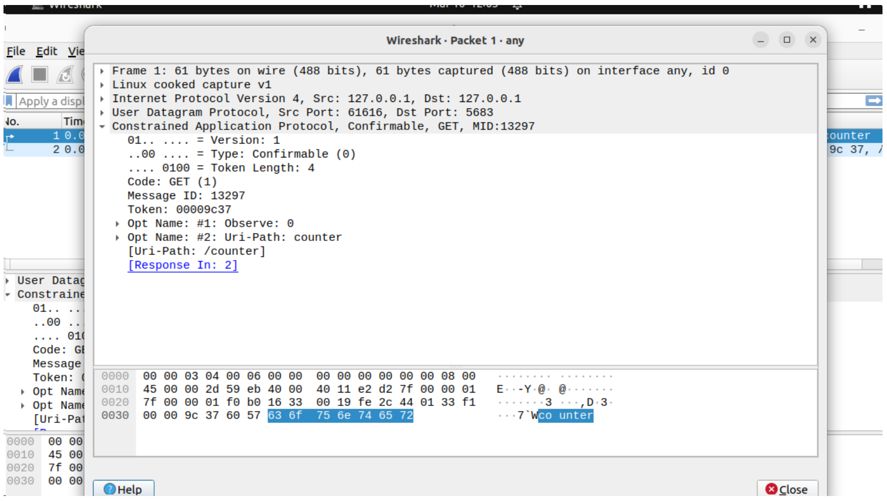
Figure 5.1: Coap Server and Client Communication



*any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter ... <Ctrl-/>

```
No.   Time          Source       Destination   Protocol  Length Info
    1 0.000000000   127.0.0.1    127.0.0.1     CoAP          61 CON, MID:13297, GET, TKN:00 00 9c 37, /counter
    2 0.000873686   127.0.0.1    127.0.0.1     CoAP          57 ACK, MID:13297, 2.05 Content, TKN:00 00 9c 37, /counte

Frame 1: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 61616, Dst Port: 5683
Constrained Application Protocol, Confirmable, GET, MID:13297

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 2d 59 eb 40 00  40 11 e2 d2 7f 00 00 01   E··-Y·@· @·······
0020  7f 00 00 01 f0 b0 16 33  00 19 fe 2c 44 01 33 f1   ·······3 ···,D·3·
0030  00 00 9c 37 60 57 63 6f  75 6e 74 65 72            ···7`Wco unter
```

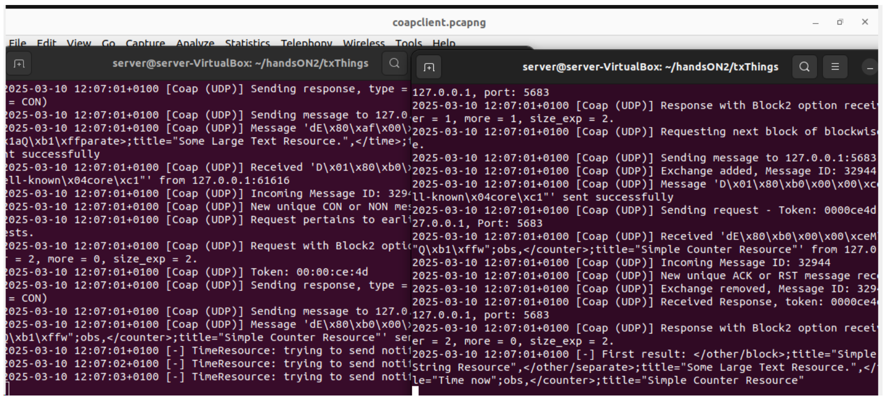
Figure 5.2: Wireshark Analysis



```
Frame 1: 61 bytes on wire (488 bits), 61 bytes captured (488 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 61616, Dst Port: 5683
Constrained Application Protocol, Confirmable, GET, MID:13297
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0100 = Token Length: 4
    Code: GET (1)
    Message ID: 13297
    Token: 00009c37
  Opt Name: #1: Observe: 0
  Opt Name: #2: Uri-Path: counter
    [Uri-Path: /counter]
    [Response In: 2]

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 2d 59 eb 40 00  40 11 e2 d2 7f 00 00 01   E··-Y·@· @·······
0020  7f 00 00 01 f0 b0 16 33  00 19 fe 2c 44 01 33 f1   ·······3 ···,D·3·
0030  00 00 9c 37 60 57 63 6f  75 6e 74 65 72            ···7`Wco unter
```

Figure 5.3: Wireshark Analysis

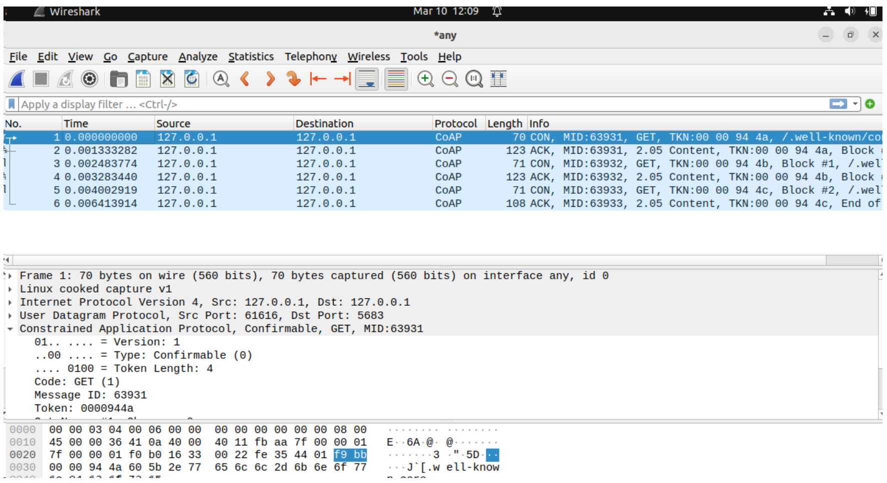
5.1.2 Running Coap Server and Coap Client ('.well-known', 'core')


coapclient.pcapng
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
server@server-VirtualBox: ~/handsON2/txThings
server@server-VirtualBox: ~/handsON2/txThings

```
2025-03-10 12:07:01+0100 [Coap (UDP)] Sending response, type =
 = CON)
2025-03-10 12:07:01+0100 [Coap (UDP)] Sending message to 127.0.
2025-03-10 12:07:01+0100 [Coap (UDP)] Message 'dE\x80\xaf\x00\x
x1aQ\xb1\xffparate>;title="Some Large Text Resource.",</time>;t
nt successfully
2025-03-10 12:07:01+0100 [Coap (UDP)] Received 'D\x01\x80\xb0\x
ell-known\x04core\xc1"' from 127.0.0.1:61616
2025-03-10 12:07:01+0100 [Coap (UDP)] Incoming Message ID: 3294
2025-03-10 12:07:01+0100 [Coap (UDP)] New unique CON or NON mes
2025-03-10 12:07:01+0100 [Coap (UDP)] Request pertains to earli
ests.
2025-03-10 12:07:01+0100 [Coap (UDP)] Request with Block2 optio
 = 2, more = 0, size_exp = 2.
2025-03-10 12:07:01+0100 [Coap (UDP)] Token: 00:00:ce:4d
2025-03-10 12:07:01+0100 [Coap (UDP)] Sending response, type =
 = CON)
2025-03-10 12:07:01+0100 [Coap (UDP)] Sending message to 127.0.
2025-03-10 12:07:01+0100 [Coap (UDP)] Message 'dE\x80\xb0\x00\x
Q\xb1\xffw";obs,</counter>;title="Simple Counter Resource"' ser
2025-03-10 12:07:01+0100 [-] TimeResource: trying to send notif
2025-03-10 12:07:02+0100 [-] TimeResource: trying to send notif
2025-03-10 12:07:03+0100 [-] TimeResource: trying to send notif
```

```
127.0.0.1, port: 5683
2025-03-10 12:07:01+0100 [Coap (UDP)] Response with Block2 option recei
er = 1, more = 1, size_exp = 2.
2025-03-10 12:07:01+0100 [Coap (UDP)] Requesting next block of blockwis
e.
2025-03-10 12:07:01+0100 [Coap (UDP)] Sending message to 127.0.0.1:5683
2025-03-10 12:07:01+0100 [Coap (UDP)] Exchange added, Message ID: 32944
2025-03-10 12:07:01+0100 [Coap (UDP)] Message 'D\x01\x80\xb0\x00\x00\xc
ll-known\x04core\xc1"' sent successfully
2025-03-10 12:07:01+0100 [Coap (UDP)] Sending request - Token: 0000ce4d
27.0.0.1, Port: 5683
2025-03-10 12:07:01+0100 [Coap (UDP)] Received 'dE\x80\xb0\x00\x00\xceM
"Q\xb1\xffw";obs,</counter>;title="Simple Counter Resource"' from 127.0
2025-03-10 12:07:01+0100 [Coap (UDP)] Incoming Message ID: 32944
2025-03-10 12:07:01+0100 [Coap (UDP)] New unique ACK or RST message rec
2025-03-10 12:07:01+0100 [Coap (UDP)] Exchange removed, Message ID: 329
2025-03-10 12:07:01+0100 [Coap (UDP)] Received Response, token: 0000ce4
127.0.0.1, port: 5683
2025-03-10 12:07:01+0100 [Coap (UDP)] Response with Block2 option recei
er = 2, more = 0, size_exp = 2.
2025-03-10 12:07:01+0100 [-] First result: </other/block>;title="Simple
String Resource",</other/separate>;title="Some Large Text Resource.",</
le="Time now";obs,</counter>;title="Simple Counter Resource"
```

Figure 5.4: Running Coap Server and Client

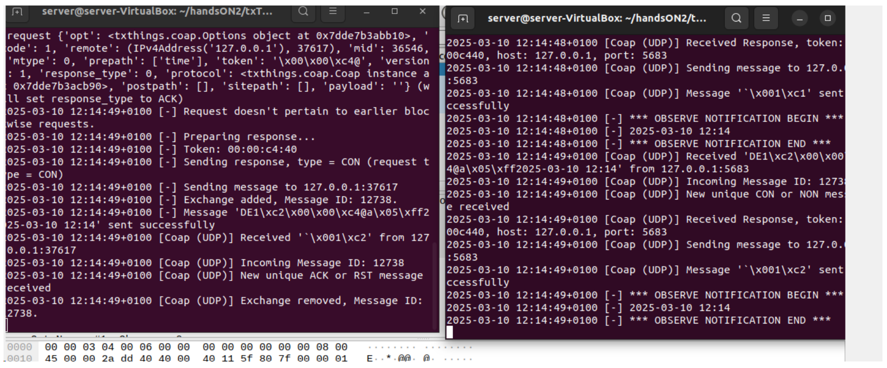

Wireshark
Mar 10 12:09
*any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.  Time         Source     Destination  Protocol  Length Info
   1 0.000000000  127.0.0.1  127.0.0.1    CoAP          70 CON, MID:63931, GET, TKN:00 00 94 4a, /.well-known/co
   2 0.001333282  127.0.0.1  127.0.0.1    CoAP         123 ACK, MID:63931, 2.05 Content, TKN:00 00 94 4a, Block
   3 0.002483774  127.0.0.1  127.0.0.1    CoAP          71 CON, MID:63932, GET, TKN:00 00 94 4b, Block #1, /.wel
   4 0.003283440  127.0.0.1  127.0.0.1    CoAP         123 ACK, MID:63932, 2.05 Content, TKN:00 00 94 4b, Block
   5 0.004002919  127.0.0.1  127.0.0.1    CoAP          71 CON, MID:63933, GET, TKN:00 00 94 4c, Block #2, /.wel
   6 0.006413914  127.0.0.1  127.0.0.1    CoAP         108 ACK, MID:63933, 2.05 Content, TKN:00 00 94 4c, End of

Frame 1: 70 bytes on wire (560 bits), 70 bytes captured (560 bits) on interface any, id 0
Linux cooked capture v1
Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
User Datagram Protocol, Src Port: 61616, Dst Port: 5683
Constrained Application Protocol, Confirmable, GET, MID:63931
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0100 = Token Length: 4
    Code: GET (1)
    Message ID: 63931
    Token: 0000944a

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 36 41 0a 40 00  40 11 fb aa 7f 00 00 01   E··6A·@· @·······
0020  7f 00 00 01 f0 b0 16 33  00 22 fe 35 44 01 f9 bb   ·······3 ·"·5D···
0030  00 00 94 4a 60 5b 2e 77  65 6c 6c 2d 6b 6e 6f 77   ···J`[.w ell-know
```

Figure 5.5: Wireshark Analysis

5.1.3 Analyze CoAP GET request/response for the `counter` resource in Wireshark

The captured traffic shows a CoAP (Constrained Application Protocol) exchange over localhost (127.0.0.1). A client sends a Confirmable (CON) GET request (MID: 36546) to retrieve the "/time" resource. The server acknowledges with an ACK and responds with 2.05 Content, returning the requested data. Some responses contain the text "Tti me", suggesting a time-related payload. Additional empty ACK messages confirm successful receptions. The interaction indicates a working local CoAP setup, likely for IoT or application testing.

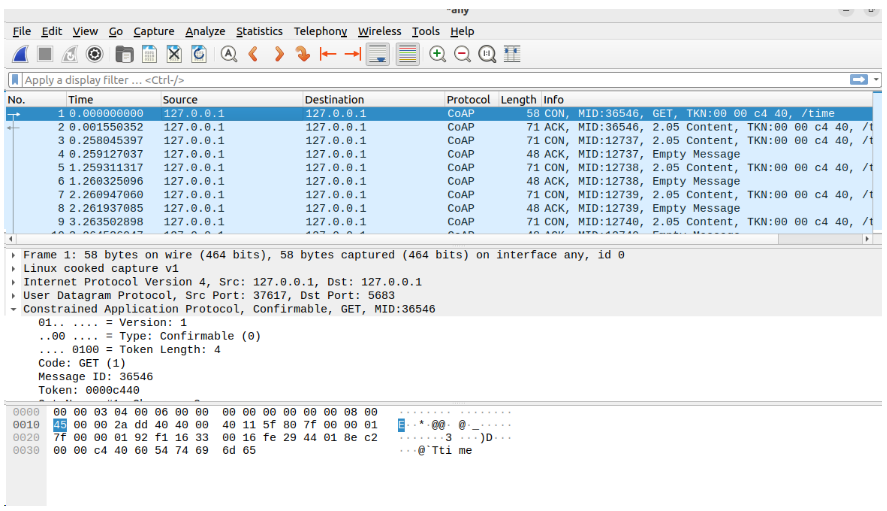

server@server-VirtualBox: ~/handsON2/txT...
server@server-VirtualBox: ~/handsON2/t...

```
request {'opt': <txthings.coap.Options object at 0x7dde7b3abb10>, '
ode': 1, 'remote': (IPv4Address('127.0.0.1'), 37617), 'mid': 36546,
'mtype': 0, 'prepath': ['time'], 'token': '\x00\x00\xc4@', 'version
: 1, 'response_type': 0, 'protocol': <txthings.coap.Coap instance a
 0x7dde7b3acb90>, 'postpath': [], 'sitepath': [], 'payload': ''} (w
ll set response_type to ACK)
025-03-10 12:14:49+0100 [-] Request doesn't pertain to earlier bloc
wise requests.
025-03-10 12:14:49+0100 [-] Preparing response...
025-03-10 12:14:49+0100 [-] Token: 00:00:c4:40
025-03-10 12:14:49+0100 [-] Sending response, type = CON (request t
pe = CON)
025-03-10 12:14:49+0100 [-] Sending message to 127.0.0.1:37617
025-03-10 12:14:49+0100 [-] Exchange added, Message ID: 12738.
025-03-10 12:14:49+0100 [-] Message 'DE1\xc2\x00\x00\xc4@a\x05\xff2
25-03-10 12:14' sent successfully
025-03-10 12:14:49+0100 [Coap (UDP)] Received '`\x001\xc2' from 127
0.0.1:37617
025-03-10 12:14:49+0100 [Coap (UDP)] Incoming Message ID: 12738
025-03-10 12:14:49+0100 [Coap (UDP)] New unique ACK or RST message
eceived
025-03-10 12:14:49+0100 [Coap (UDP)] Exchange removed, Message ID:
2738.
```

```
2025-03-10 12:14:48+0100 [Coap (UDP)] Received Response, token:
00c440, host: 127.0.0.1, port: 5683
2025-03-10 12:14:48+0100 [Coap (UDP)] Sending message to 127.0.
:5683
2025-03-10 12:14:48+0100 [Coap (UDP)] Message '`\x001\xc1' sent
ccessfully
2025-03-10 12:14:48+0100 [-] *** OBSERVE NOTIFICATION BEGIN ***
2025-03-10 12:14:48+0100 [-] 2025-03-10 12:14
2025-03-10 12:14:48+0100 [-] *** OBSERVE NOTIFICATION END ***
2025-03-10 12:14:49+0100 [Coap (UDP)] Received 'DE1\xc2\x00\x00
4@a\x05\xff2025-03-10 12:14' from 127.0.0.1:5683
2025-03-10 12:14:49+0100 [Coap (UDP)] Incoming Message ID: 1273
2025-03-10 12:14:49+0100 [Coap (UDP)] New unique CON or NON mes
e received
2025-03-10 12:14:49+0100 [Coap (UDP)] Received Response, token:
00c440, host: 127.0.0.1, port: 5683
2025-03-10 12:14:49+0100 [Coap (UDP)] Sending message to 127.0.
:5683
2025-03-10 12:14:49+0100 [Coap (UDP)] Message '`\x001\xc2' sent
ccessfully
2025-03-10 12:14:49+0100 [-] *** OBSERVE NOTIFICATION BEGIN ***
2025-03-10 12:14:49+0100 [-] 2025-03-10 12:14
2025-03-10 12:14:49+0100 [-] *** OBSERVE NOTIFICATION END ***
```

```
0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 2a dd 40 40 00  40 11 5f 80 7f 00 00 01   E··*·@@· @·_·····
```

Figure 5.6: Running Coap Clientobserve


*any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
Apply a display filter … <Ctrl-/>

```
No.  Time          Source        Destination   Protocol  Length  Info
   1 0.000000000   127.0.0.1     127.0.0.1     CoAP        58 CON, MID:36546, GET, TKN:00 00 c4 40, /time
   2 0.001550352   127.0.0.1     127.0.0.1     CoAP        71 ACK, MID:36546, 2.05 Content, TKN:00 00 c4 40, /t
   3 0.258045397   127.0.0.1     127.0.0.1     CoAP        71 CON, MID:12737, 2.05 Content, TKN:00 00 c4 40, /t
   4 0.259127037   127.0.0.1     127.0.0.1     CoAP        48 ACK, MID:12737, Empty Message
   5 1.259311317   127.0.0.1     127.0.0.1     CoAP        71 CON, MID:12738, 2.05 Content, TKN:00 00 c4 40, /t
   6 1.260325096   127.0.0.1     127.0.0.1     CoAP        48 ACK, MID:12738, Empty Message
   7 2.260947060   127.0.0.1     127.0.0.1     CoAP        71 CON, MID:12739, 2.05 Content, TKN:00 00 c4 40, /t
   8 2.261937085   127.0.0.1     127.0.0.1     CoAP        48 ACK, MID:12739, Empty Message
   9 3.263502898   127.0.0.1     127.0.0.1     CoAP        71 CON, MID:12740, 2.05 Content, TKN:00 00 c4 40, /t
```

```
▸ Frame 1: 58 bytes on wire (464 bits), 58 bytes captured (464 bits) on interface any, id 0
▸ Linux cooked capture v1
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 37617, Dst Port: 5683
▾ Constrained Application Protocol, Confirmable, GET, MID:36546
    01.. .... = Version: 1
    ..00 .... = Type: Confirmable (0)
    .... 0100 = Token Length: 4
    Code: GET (1)
    Message ID: 36546
    Token: 0000c440
```

```
0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ········ ········
0010  45 00 00 2a dd 40 40 00  40 11 5f 80 7f 00 00 01   E··*·@@· @·_·····
0020  7f 00 00 01 92 f1 16 33  00 16 fe 29 44 01 8e c2   ·······3 ···)D···
0030  00 00 c4 40 60 54 74 69  6d 65                     ···@`Tti me
```

Figure 5.7: Wireshark Analysis

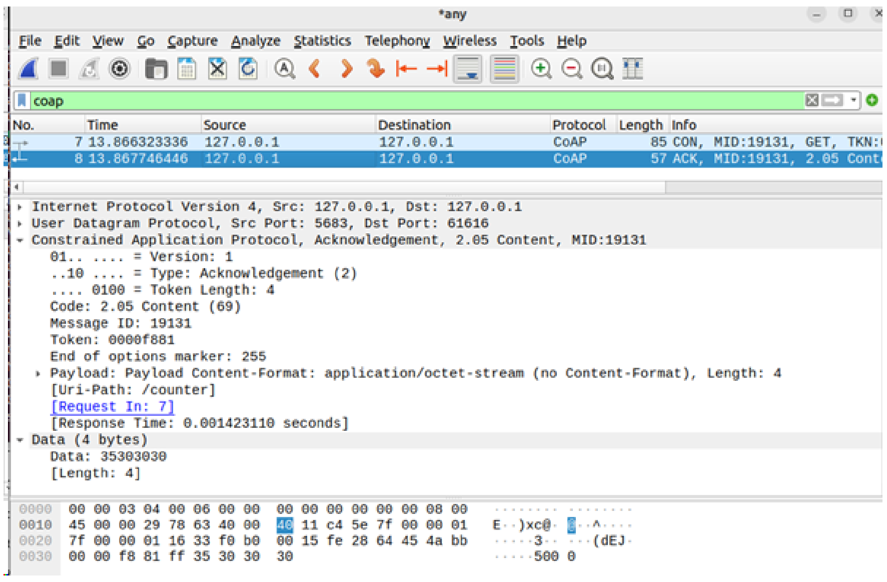
5.1.4 Security Analysis Results

The implementation was validated through multiple test scenarios:

- **Authorized Request:** When a client transmitted a request containing the correct authentication token (s3cr3t), encrypted using the agreed-upon AES key and IV, the server successfully decrypted the payload and authenticated the request. The server then responded with the current value of the counter resource e.g., `5000`, demonstrating that the secure communication channel was functioning as intended
- **Unauthorized Request:**

  If a client sent a request with an invalid token or tampered encrypted payload, the server responded with a 401 UNAUTHORIZED CoAP response code, accompanied by an appropriate error message. This confirms that the server correctly rejected unauthenticated or corrupted requests without processing them further.

Sample server output:


*any
File Edit View Go Capture Analyze Statistics Telephony Wireless Tools Help
coap

```
No.   Time          Source      Destination  Protocol  Length  Info
    7 13.866323336  127.0.0.1   127.0.0.1    CoAP          85 CON, MID:19131, GET, TKN:
    8 13.867746446  127.0.0.1   127.0.0.1    CoAP          57 ACK, MID:19131, 2.05 Cont

▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 5683, Dst Port: 61616
▾ Constrained Application Protocol, Acknowledgement, 2.05 Content, MID:19131
    01.. .... = Version: 1
    ..10 .... = Type: Acknowledgement (2)
    .... 0100 = Token Length: 4
    Code: 2.05 Content (69)
    Message ID: 19131
    Token: 0000f881
    End of options marker: 255
  ▸ Payload: Payload Content-Format: application/octet-stream (no Content-Format), Length: 4
    [Uri-Path: /counter]
    [Request In: 7]
    [Response Time: 0.001423110 seconds]
▾ Data (4 bytes)
    Data: 35303030
    [Length: 4]

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ........ ........
0010  45 00 00 29 78 63 40 00  40 11 c4 5e 7f 00 00 01   E··)xc@· @··^····
0020  7f 00 00 01 16 33 f0 b0  00 15 fe 28 64 45 4a bb   ·····3·· ···(dEJ·
0030  00 00 f8 81 ff 35 30 30  30                        ·····500 0
```

Figure 5.8: Authorized Access

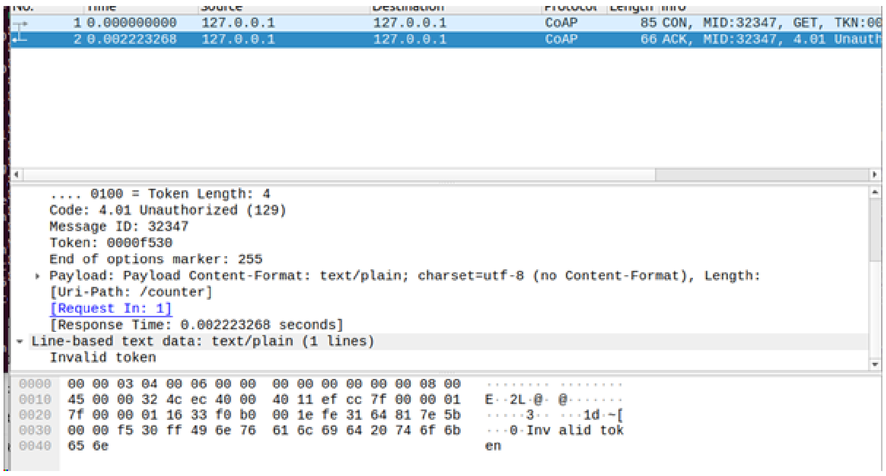
```
First result: 5000
```

Sample output for unauthorized access:

```
First result: Invalid token.
```

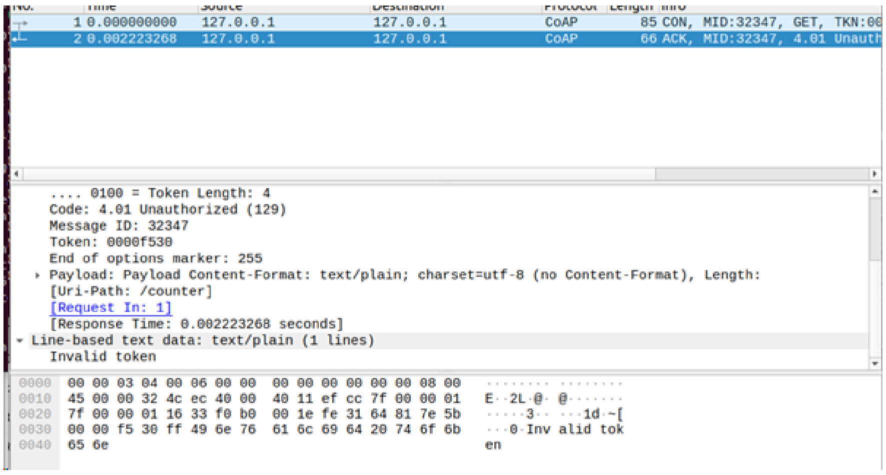
Traffic analysis using Wireshark confirmed that:

- Unlike previous implementations, payload content was no longer visible in cleartext within UDP datagrams. Encrypted tokens appeared as unintelligible binary data, confirming successful AES encryption.
- No signs of packet tampering or truncation were observed. The server consistently responded with the expected CoAP codes (2.05 Content for valid requests, 4.01 Unauthorized for invalid ones).
- Unauthorized requests did not result in any changes to server-side state (e.g., counter value), and were terminated immediately after payload decryption and token verification.



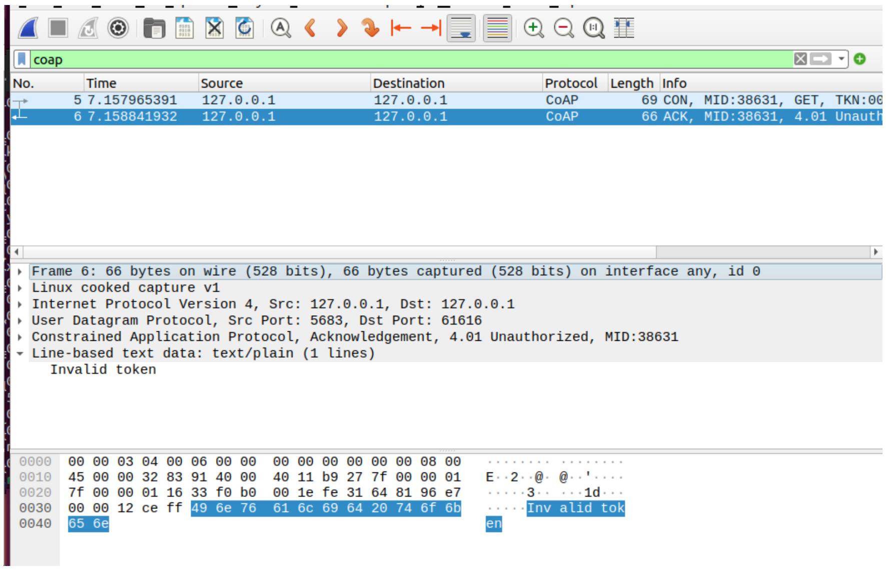
```
   1 0.000000000   127.0.0.1      127.0.0.1      CoAP      85 CON, MID:32347, GET, TKN:00
   2 0.002223268   127.0.0.1      127.0.0.1      CoAP      66 ACK, MID:32347, 4.01 Unauth

    .... 0100 = Token Length: 4
    Code: 4.01 Unauthorized (129)
    Message ID: 32347
    Token: 0000f530
    End of options marker: 255
  ▸ Payload: Payload Content-Format: text/plain; charset=utf-8 (no Content-Format), Length:
    [Uri-Path: /counter]
    [Request In: 1]
    [Response Time: 0.002223268 seconds]
▾ Line-based text data: text/plain (1 lines)
    Invalid token

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ........ ........
0010  45 00 00 32 4c ec 40 00  40 11 ef cc 7f 00 00 01   E··2L·@· @·······
0020  7f 00 00 01 16 33 f0 b0  00 1e fe 31 64 81 7e 5b   ·····3·· ···1d·~[
0030  00 00 f5 30 ff 49 6e 76  61 6c 69 64 20 74 6f 6b   ···0·Inv alid tok
0040  65 6e                                              en
```

Figure 5.9: Unauthorized Access



```
   1 0.000000000   127.0.0.1      127.0.0.1      CoAP      85 CON, MID:32347, GET, TKN:00
   2 0.002223268   127.0.0.1      127.0.0.1      CoAP      66 ACK, MID:32347, 4.01 Unauth

    .... 0100 = Token Length: 4
    Code: 4.01 Unauthorized (129)
    Message ID: 32347
    Token: 0000f530
    End of options marker: 255
  ▸ Payload: Payload Content-Format: text/plain; charset=utf-8 (no Content-Format), Length:
    [Uri-Path: /counter]
    [Request In: 1]
    [Response Time: 0.002223268 seconds]
▾ Line-based text data: text/plain (1 lines)
    Invalid token

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ........ ........
0010  45 00 00 32 4c ec 40 00  40 11 ef cc 7f 00 00 01   E··2L·@· @·······
0020  7f 00 00 01 16 33 f0 b0  00 1e fe 31 64 81 7e 5b   ·····3·· ···1d·~[
0030  00 00 f5 30 ff 49 6e 76  61 6c 69 64 20 74 6f 6b   ···0·Inv alid tok
0040  65 6e                                              en
```

Figure 5.10: Wireshark Inspect



```
coap
No.  Time          Source         Destination    Protocol  Length Info
   5 7.157965391   127.0.0.1      127.0.0.1      CoAP      69 CON, MID:38631, GET, TKN:00
   6 7.158841932   127.0.0.1      127.0.0.1      CoAP      66 ACK, MID:38631, 4.01 Unauth

▸ Frame 6: 66 bytes on wire (528 bits), 66 bytes captured (528 bits) on interface any, id 0
▸ Linux cooked capture v1
▸ Internet Protocol Version 4, Src: 127.0.0.1, Dst: 127.0.0.1
▸ User Datagram Protocol, Src Port: 5683, Dst Port: 61616
▸ Constrained Application Protocol, Acknowledgement, 4.01 Unauthorized, MID:38631
▾ Line-based text data: text/plain (1 lines)
    Invalid token

0000  00 00 03 04 00 06 00 00  00 00 00 00 00 00 08 00   ........ ........
0010  45 00 00 32 83 91 40 00  40 11 b9 27 7f 00 00 01   E··2··@· @··'····
0020  7f 00 00 01 16 33 f0 b0  00 1e fe 31 64 81 96 e7   ·····3·· ···1d···
0030  00 00 12 ce ff 49 6e 76  61 6c 69 64 20 74 6f 6b   ·····Inv alid tok
0040  65 6e                                              en
```

Figure 5.11: Unauthorized Access

6

Discussion

6.1 System Behavior and Analysis

This section reflects on the practical aspects of implementing a CoAP-based communication system, focusing on system behavior, traffic analysis, and performance insights observed during experimentation.

6.1.1 CoAP Server and Client Behavior

The CoAP server (`coap_server.py`) was successfully deployed on both a Raspberry Pi and a Linux virtual machine. The server exposed a time-based resource which clients could query or observe. On the client side, interactions were initiated using a Python script that issued GET requests and set up observation for real-time updates.

Testing showed reliable communication over the default CoAP port (5683) using the UDP transport layer. The server responded with valid `2.05 Content` messages upon receiving a GET, confirming correct handling of CoAP requests. Observations triggered timely notifications when the server-side resource changed, demonstrating CoAP's built-in support for asynchronous updates.

6.1.2 Wireshark Traffic Analysis

Wireshark was used to capture and analyze traffic between the client and server. Clear CoAP message exchanges were visible, with key packet types such as Confirmable (CON), Acknowledgement (ACK), and Reset (RST). In one trace, a client sent a GET request for the `/time` resource, which the server responded to with a `2.05 Content` message—containing a textual payload representing the current time.

Figures 5.1 through 5.7 highlight these exchanges. For example, Figure 5.5 illustrates a Confirmable GET request followed by a corresponding ACK response. Observed traffic confirmed the lightweight nature of CoAP, as messages were minimal in size and lacked the overhead typical of protocols like HTTP.

6.1.3 Observations and System Responsiveness

Observation mode, implemented via `clientobserve.py`, allowed the client to automatically receive updates when the resource changed, without needing to poll the server repeatedly. This feature greatly enhances the efficiency of IoT systems where devices have limited processing and energy capacity.

The system's responsiveness was high under normal network conditions, with low latency between the client's request and the server's response. Even on constrained hardware like the Raspberry Pi, the server maintained stable performance, indicating the suitability of CoAP for low-power environments.

6.1.4 Security Analysis

The integration of both token-based authentication and AES encryption into the CoAP communication workflow significantly enhances the security posture of the system. This dual-layered approach provides not only access control but also ensures the confidentiality of transmitted data. Token-based validation enforces a minimal authentication mechanism, while AES encryption protects message payloads against eavesdropping and passive attacks.

This security design is particularly suited for constrained environments where the overhead of standardized protocols such as DTLS or OSCORE may be impractical. The use of symmetric key cryptography ensures that the added security does not impose substantial computational or memory overhead, thus preserving the lightweight nature of the CoAP protocol.

Unlike the initial implementation which transmitted tokens in plaintext, the enhanced version prevents exposure of authentication credentials by encrypting the payload prior to transmission. This improvement mitigates the risk of credential leakage and reinforces the integrity and privacy of the communication.

Potential improvements to this approach include:

- Introducing time-bound or rotating tokens for session-level security.
- Incorporating client-specific validation, such as binding tokens to IP addresses or device identifiers.

The current implementation represents a practical compromise between security and efficiency, aligning with the foundational objectives of IoT protocol design. It serves as a robust starting point for secure CoAP-based communication in resource-constrained deployments, with room for incremental enhancements as needed.

7 Conclusion

7.1 Conclusion

This project demonstrated the implementation and evaluation of a CoAP-based communication system tailored for IoT environments. By deploying the CoAP server on both a Raspberry Pi and a Linux virtual machine, and interacting with it using a Python-based client, the system effectively showcased lightweight communication over UDP.

The implementation successfully handled CoAP GET requests and supported resource observation, allowing real-time data updates without requiring frequent polling. This confirmed the protocol's efficiency and responsiveness in constrained environments.

Packet-level analysis using Wireshark validated the correct operation of the protocol, highlighting its minimal overhead and confirming message exchanges such as Confirmable requests, Acknowledgements, and Content responses. The system performed reliably under different test conditions and maintained stable communication even on low-power hardware.

In addition to core functionality, a lightweight token-based security feature was incorporated into the CoAP messaging process. By embedding a pre-shared authentication token within the payload, the server was able to restrict access to authorized clients only. This mechanism provided a simple and effective layer of access control without introducing significant computational overhead or altering the CoAP protocol structure. Unauthorized requests were promptly rejected, ensuring the integrity of the system and enhancing its suitability for real-world deployment where basic security enforcement is necessary.

Overall, the project highlights CoAP's practicality for IoT applications that demand low-latency, low-overhead communication. Its simplicity, resource awareness, support for observation, and compatibility with lightweight security enhancements make it a strong candidate for secure and efficient deployment in various IoT scenarios.

References

[1] Markel Iglesias-Urkia, Adrián Orive, and Aitor Urbieta. Analysis of coap implementations for industrial internet of things: A survey. *Procedia Computer Science*, 109:188–195, 2017.

[2] Digvijaysinh Rathod and Sunit Patil. Security analysis of constrained application protocol (coap): Iot protocol. *International Journal of Advanced Studies in Computers, Science and Engineering*, 6(8):37, 2017.

[3] Reem Abdul Rahman and Babar Shah. Security analysis of iot protocols: A focus in coap. In *2016 3rd MEC international conference on big data and smart city (ICBDSC)*, pages 1–7. IEEE, 2016.

[4] Björsn Konieczek, Michael Rethfeldt, Frank Golatowski, and Dirk Timmermann. Real-time communication for the internet of things using jcoap. In *2015 IEEE 18th International Symposium on Real-Time Distributed Computing*, pages 134–141. IEEE, 2015.

[5] Sharwari Satish Solapure and Harish H Kenchannavar. Rpl and coap protocols, experimental analysis for iot: A case study. *International Journal of Ad hoc, Sensor & Ubiquitous Computing (IJASUC)*, 10(2), 2019.

[6] Seung-Man Chun, Hyun-Su Kim, and Jong-Tae Park. Coap-based mobility management for the internet of things. *Sensors*, 15(7):16060–16082, 2015.