



Implementation and Analysis of a CoAP-Based IoT Communication System

Mohammed Alghubari - Majeed Alkhanferi - Mamdouh Alzahrani

202307030 - 202307010 - 202307270

King Fahd University
April 2025

Contents

	Page
<i>List of Figures</i>	<i>III</i>
<i>List of Tables</i>	<i>IV</i>
1 Abstract	1
2 Introduction	2
2.1 Background Introduction	2
2.1.1 Problem Statement	2
2.1.2 Aim and Objectives	2
3 Methodology	4
3.0.1 Approach Overview	4
3.0.2 Tools and Environment	4
3.1 Tools and Environment	4
3.1.1 Hardware	4
3.1.2 Software	4
3.1.3 Network Configuration	5
4 Implementation	6
4.1 Adding Security To Coap	17
4.2 Client-Side Implementation	17
4.3 Server-Side Verification	17
5 Result	18
5.1 Capture Packet Using Wireshark	18
5.1.1 Running Coap Server and Client	18
5.1.2 Running Coap Server and Coap Client ('well-known', 'core')	19
5.1.3 Analyze CoAP GET request/response for the counter resource in Wireshark	20
5.1.4 Security Analysis Results	21
6 Discussion	23
6.1 System Behavior and Analysis	23
6.1.1 CoAP Server and Client Behavior	23
6.1.2 Wireshark Traffic Analysis	23
6.1.3 Observations and System Responsiveness	23
6.1.4 Security Analysis	23

7 Conclusion	25
7.1 Conclusion	25
<i>References</i>	<i>26</i>

List of Figures

	Page
3.1 A SIMPLE RASPBERRY PI SETUP	5
4.1	14
4.2	14
4.3 Wireshark Analysis of Coap Communication between Linux and Raspberry Pi	16
5.1 Coap Server and Client Communication	18
5.2 Wireshark Analysis	18
5.3 Wireshark Analysis	19
5.4 Running Coap Server and Client	19
5.5 Wireshark Analysis	19
5.6 Running Coap Clientobserve	20
5.7 Wireshark Analysis	20
5.8 Authorized Access	21
5.9 Unauthorized Access	21
5.10 Unauthorized Access	22

List of Tables

Page

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.

2.1 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.

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

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)

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

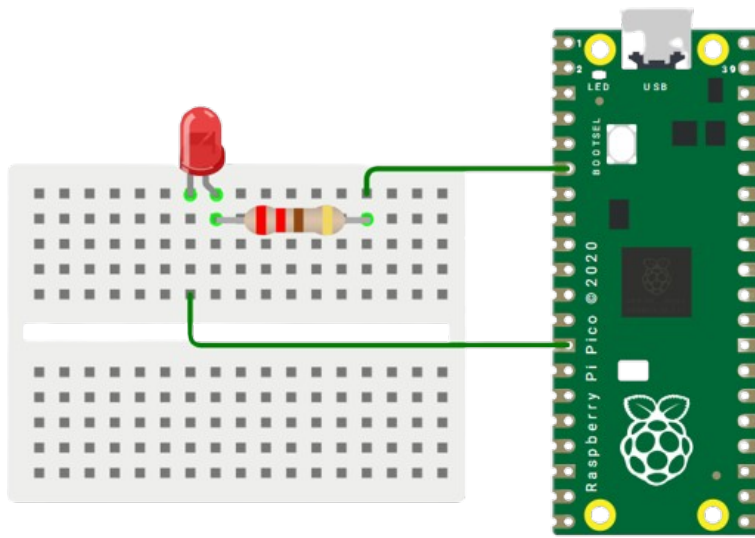


Figure 3.1: A SIMPLE RASPBERRY PI SETUP

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 CoAPResource 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 programming. There are two type of callbacks: normal callbacks and errbacks, which handle an exception. 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 IoT
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 to"
    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("%Y-%m-%d %H:%M:%S"))
    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 default 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 performed.

    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: Viewing Observable Resources in Chrome

Install [Copper extension for Chrome](#). Browse `coap://<server-ip>:5683/time` and click Observe.

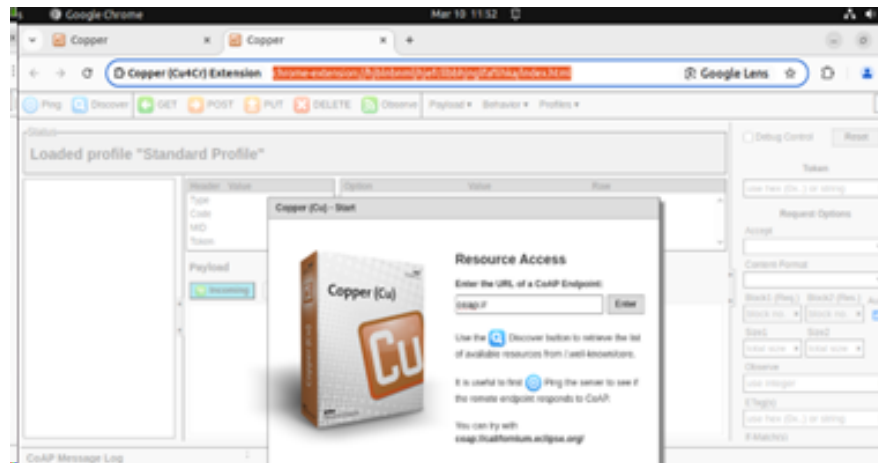


Figure 4.1

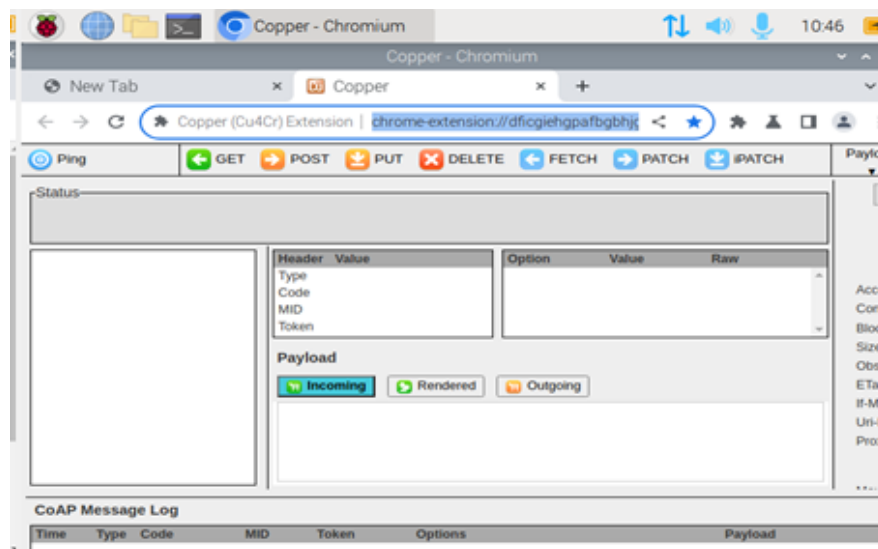


Figure 4.2

Step 8: 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 Raspberry Pi Vm and Coap Client in Linux VM

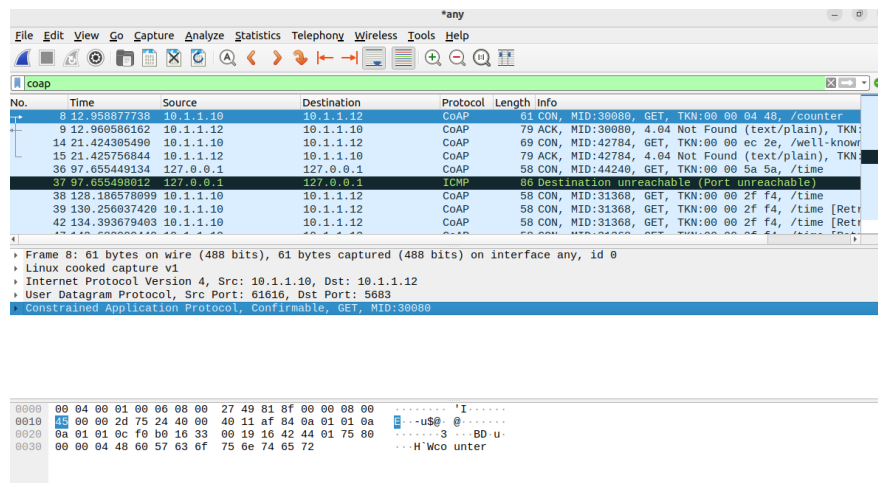


Figure 4.3: Wireshark Analysis of Coap Communication between Linux and Raspberry Pi

4.1 Adding Security To Coap

To ensure access control and restrict unauthorized requests, a shared secret token was embedded into the CoAP message payload. This token acts as a pre-shared key (PSK) known to both the client and the server. The mechanism does not alter the core CoAP protocol but instead leverages its payload capability for authentication.

4.2 Client-Side Implementation

The client application constructs a CoAP GET request targeting the `counter` resource. The authentication token is appended to the request payload as follows:

```
request = coap.Message(code=coap.GET)
request.opt.uri_path = ['counter']
request.opt.observe = 0
request.payload = "s3cr3t"
request.remote = (ip_address("127.0.0.1"), coap.COAP_PORT)
```

This approach ensures that every request includes the authentication string without modifying CoAP headers or options.

4.3 Server-Side Verification

The server, upon receiving a GET request for the `counter` resource, performs a verification check on the payload. If the token does not match the predefined value, the server responds with an `UNAUTHORIZED` CoAP response code:

```
if request.payload != "s3cr3t":
    response = coap.Message(code=coap.UNAUTHORIZED, payload="Invalid_tok
    return defer.succeed(response)
```

Only requests containing the correct token are processed further, and the counter value is returned as a response.

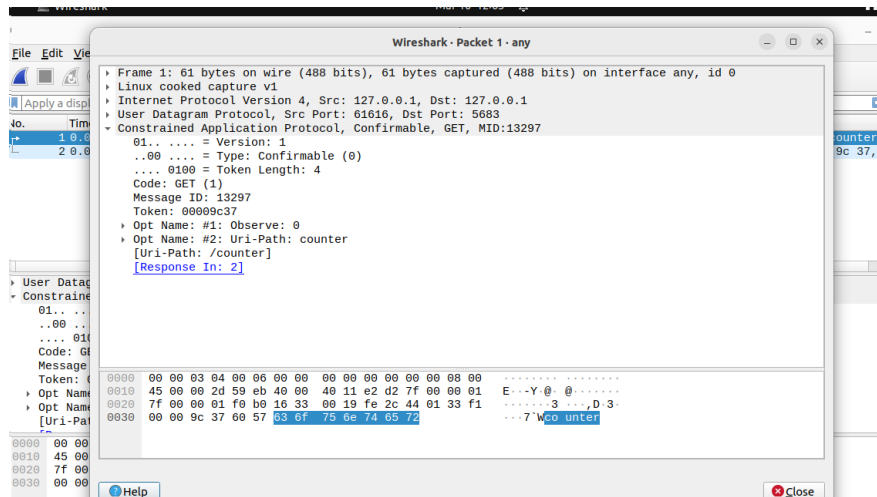


Figure 5.3: Wireshark Analysis

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

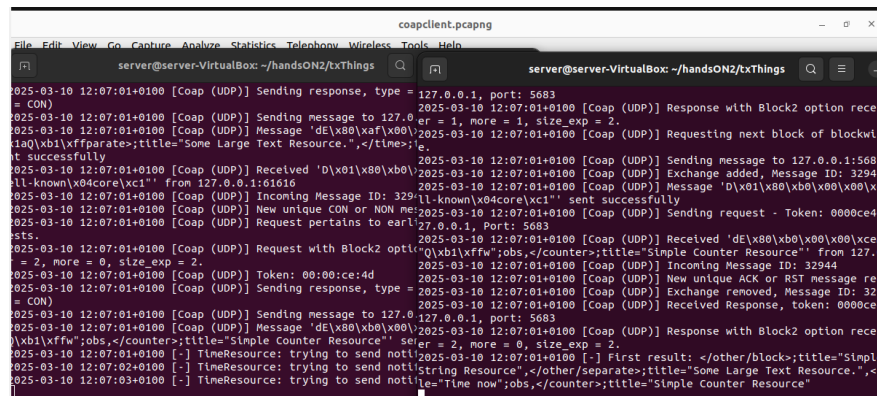


Figure 5.4: Running Coap Server and Client

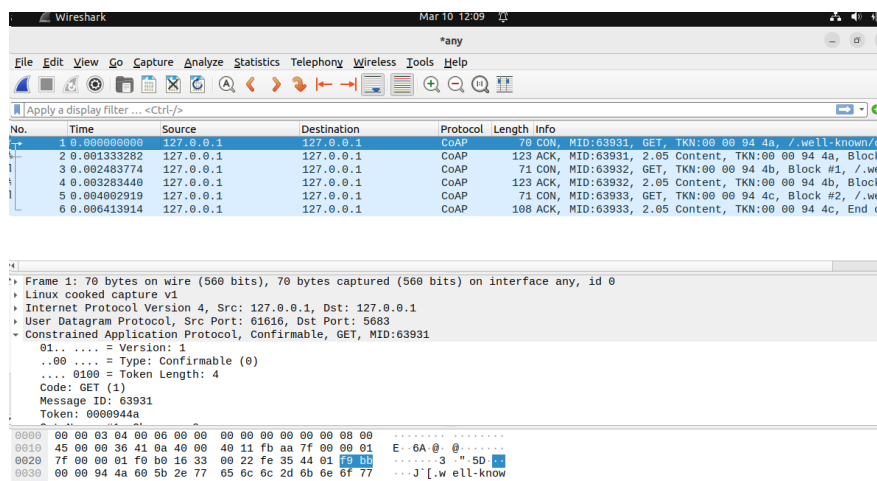


Figure 5.5: Wireshark Analysis

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/rxt...
request {'get': <txthings.coap.Options object at 0x7dde7b3abb10>,
'code': 1, 'remote': (IPv4Address('127.0.0.1'), 37617), 'mid': 36546,
'ntype': 0, 'prepath': ['tme'], 'token': '\x00\x00\xc4', 'version':
1, 'response_type': 0, 'protocol': <txthings.coap.Coap instance at
0x7dde7b3ac990>, 'postpath': [], 'slstepath': [], 'payload': ''} (w
ill set response type to ACK)
2025-03-10 12:14:49+0100 [-] Request doesn't pertain to earlier bloc
wise requests.
2025-03-10 12:14:49+0100 [-] Preparing response...
2025-03-10 12:14:49+0100 [-] Token: 00:00:c4:40
2025-03-10 12:14:49+0100 [-] Sending response, type = CON (request t
pe = CON)
2025-03-10 12:14:49+0100 [-] Sending message to 127.0.0.1:37617
2025-03-10 12:14:49+0100 [-] Exchange added, Message ID: 12738.
2025-03-10 12:14:49+0100 [-] Message 'DE1\xc2\x00\x00\xc4a\x05\xff2
25-03-10 12:14' sent successfully
2025-03-10 12:14:49+0100 [Coap (UDP)] Received '\x001\xc2' from 127
0.0.1:37617
2025-03-10 12:14:49+0100 [Coap (UDP)] Incoming Message ID: 12738
2025-03-10 12:14:49+0100 [Coap (UDP)] New unique ACK or RST message
received
2025-03-10 12:14:49+0100 [Coap (UDP)] Exchange removed, Message ID:
2738.
0000 00 00 03 04 00 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010 45 00 00 2a dd 40 00 00 40 11 5f 80 7f 00 00 01 E:*.@. @. ....

```

Figure 5.6: Running Coap Client to observe

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	0.000155952	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	2.261937885	127.0.0.1	127.0.0.1	CoAP	48	ACK, MID:12739, Empty Message
9	3.263592890	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 00 00 00 00 00 00 00 00 00 00 00 00 00 00
0010 45 00 00 2a dd 40 00 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 c23...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 the client sends the correct token (`s3cr3t`), the server responds with the current counter value, e.g., 5000.
- **Unauthorized Request:** If the token is missing or incorrect, the server replies with an error message indicating unauthorized access.

Sample server output:

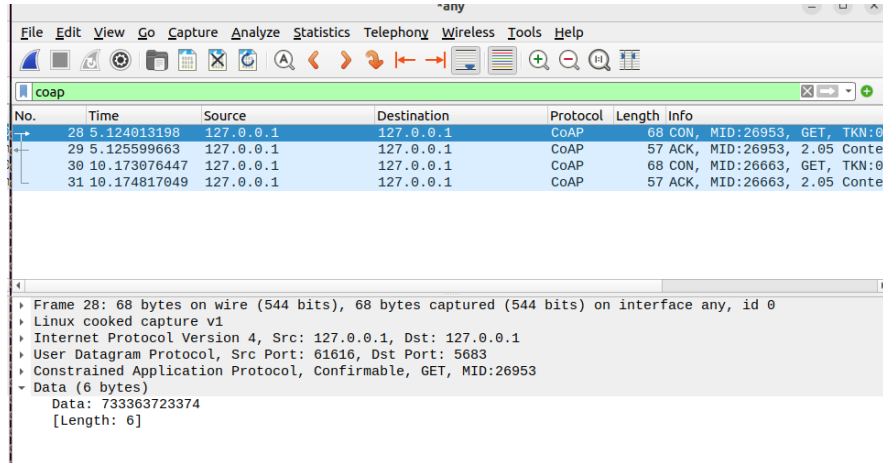


Figure 5.8: Authorized Access

First result : 5000

Sample output for unauthorized access:

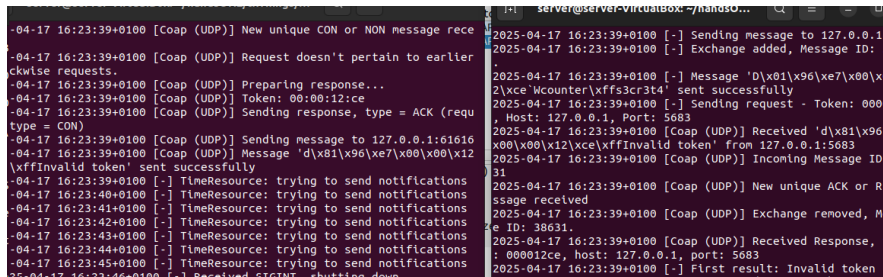


Figure 5.9: Unauthorized Access

First result : Invalid token.

Traffic analysis using Wireshark confirmed that:

- The payload was transmitted in plaintext, visible within the UDP datagram.
- Message integrity was maintained, and the response codes matched the expected behavior.
- Unauthorized messages did not trigger state changes or processing on the server beyond token verification.

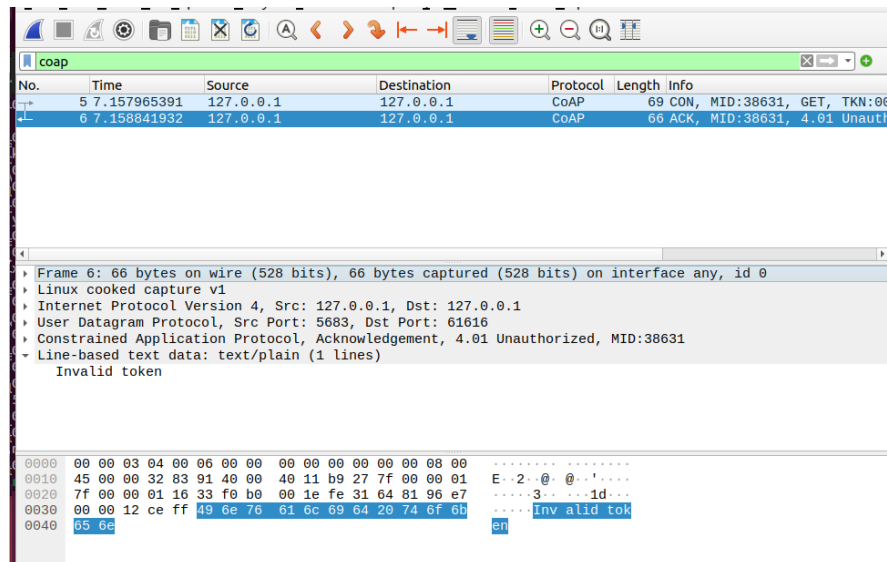


Figure 5.10: Unauthorized Access

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 token-based authentication into the CoAP workflow provides a basic but effective access control mechanism. This method is particularly suitable for lightweight deployments where the overhead of DTLS or OSCORE is undesirable or unsupported.

While the approach does not offer message encryption or protection against eavesdropping or replay attacks, it establishes a minimal authentication layer that can deter casual

unauthorized access. Additionally, its simplicity ensures compatibility with existing CoAP infrastructure and minimal impact on performance.

Potential improvements to this approach include:

- Introducing time-bound or rotating tokens for session-level security.
- Using signed or hashed payloads to prevent tampering.
- Extending token validation with client-specific identifiers or IP address checks.

The current implementation offers a practical compromise between security and resource constraints, aligning with the core principles of IoT protocol design.

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 Urbieto. 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] Björns Konieczek, Michael Rethfeldt, Frank Glatowski, 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.