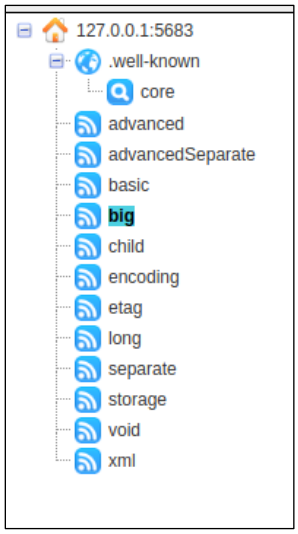
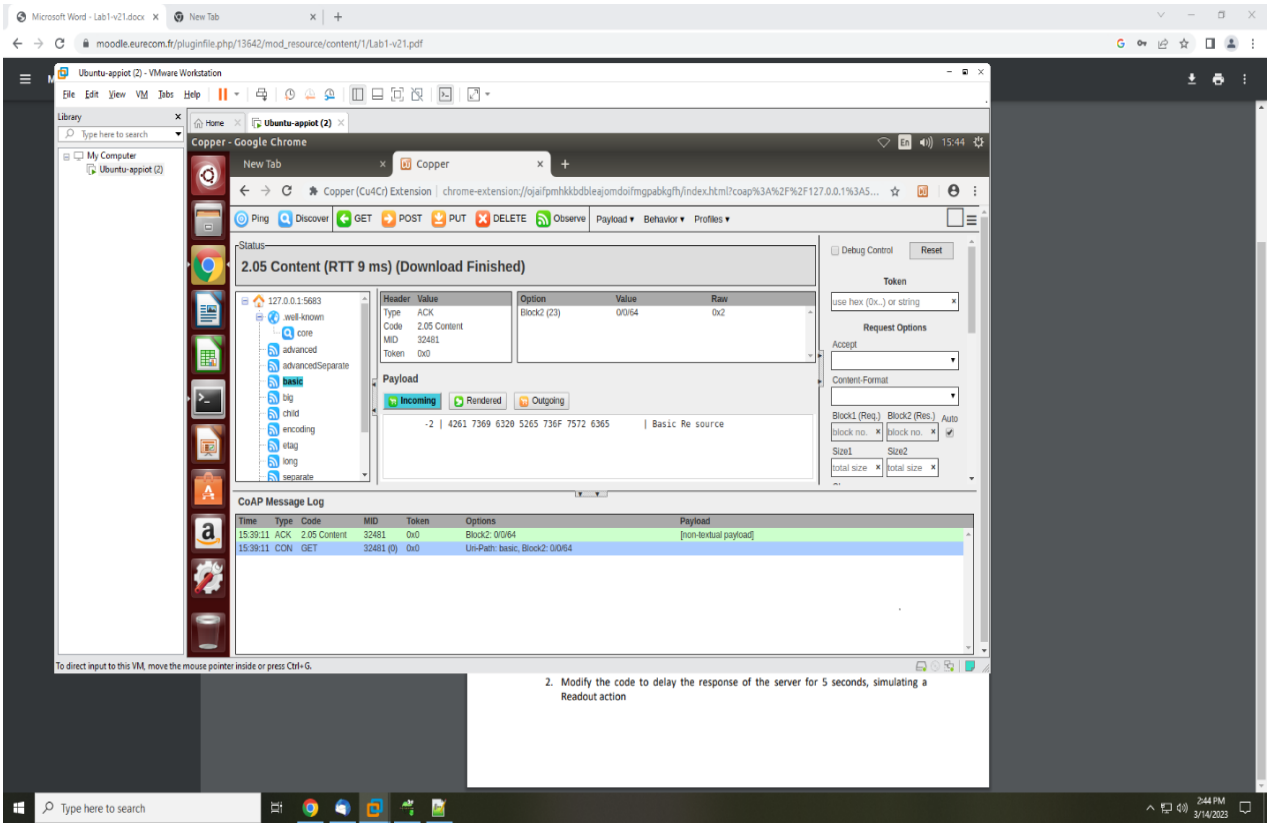
LAB 1: COAP

Shubhika GARG

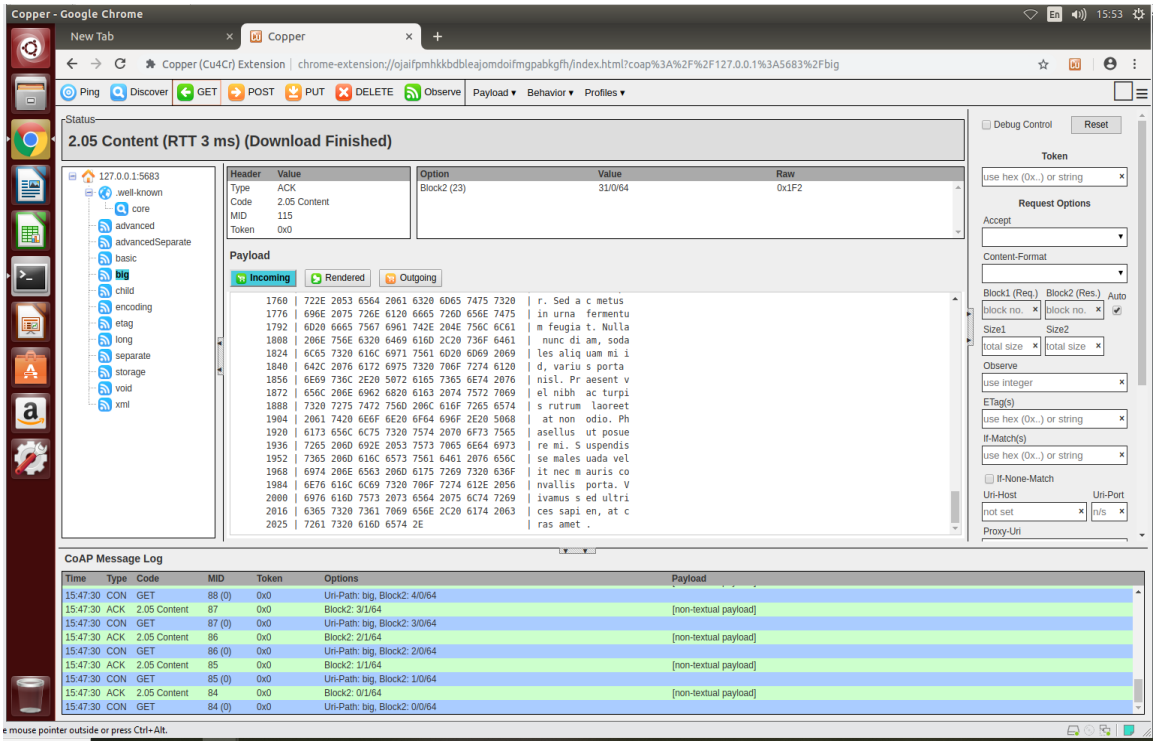
- **How to know the available resources at the server? Try to do it with COAP. What is the result?**
  On Chrome, on clicking on the COPPER extension, and entering "coap://127.0.0.1:5683", I was able to see the resources of the server on the left window. For example, advanced, advancedSeparate, basic, big, child, encoding, etag, long, separate, storage, void and xml are the available resources in the coap server.
  The available resources are hosted by Coapthon, where the coapserver.py runs on the terminal. We can retrieve and modify the available resources using the standard requests (GET, PUT, POST and DELETE) with the help of the coap client.



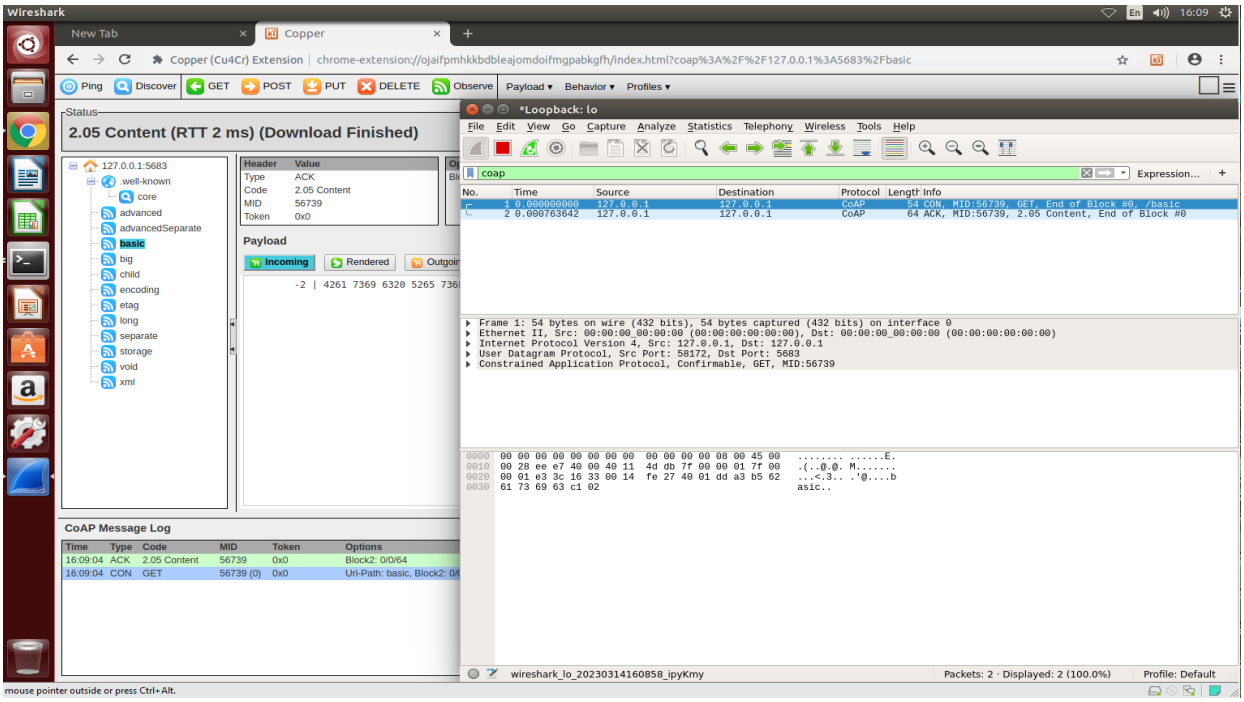
- **Select "basic" on right and click GET. What happened?**
  When a GET Request is made on selecting a Basic Resource, the server's resource tree looks for the resource requested by the client and then the associated handler executes the method requested by COAP. The Request/Response approach used by the COAP is similar to that of HTTP. The basic resource will enable the reliable delivery of the message implemented over the UDP. Now, since the reliable delivery is enabled the request/response are transmitted inside the CON message (confirmable) type, it needs an acknowledgement (ACK) message from the receiver. The acknowledgement is delivered in a different mode, so as soon as the request is received by the clients, the receiver instantly sends an empty ACK message and when the response is ready, the actual CON message is sent.

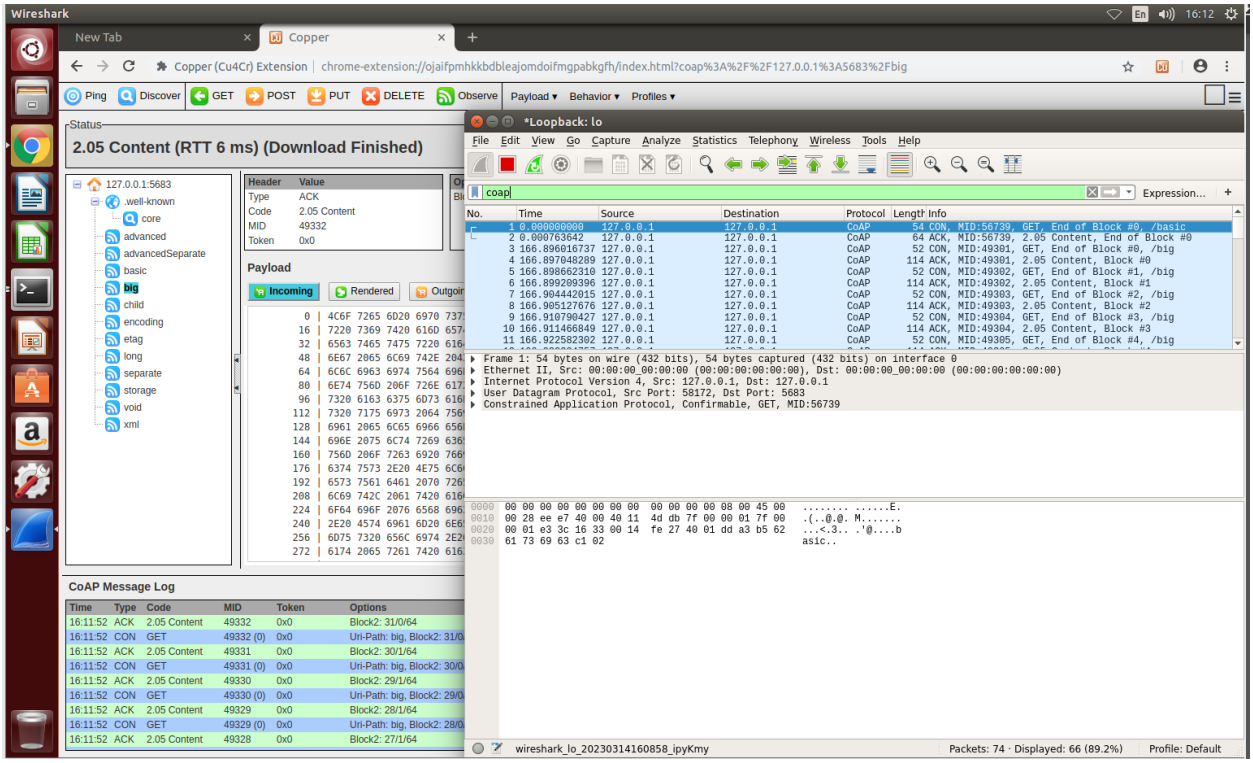


- **Repeat with "big". What happened?**
  Using big resource, a large amount of data is transferred between the client and the server, In COAP there is an extension feature called Block-wise transfer that allows large data to be fragmented directly in the application in the format of message chain to avoid IP fragmentation. For example, it is transferred into several blocks and enables the reliable delivery of the message implemented over the UDP by sending an empty acknowledgement ACK message after the receiver received a request and then sending the actual response in the CON message.

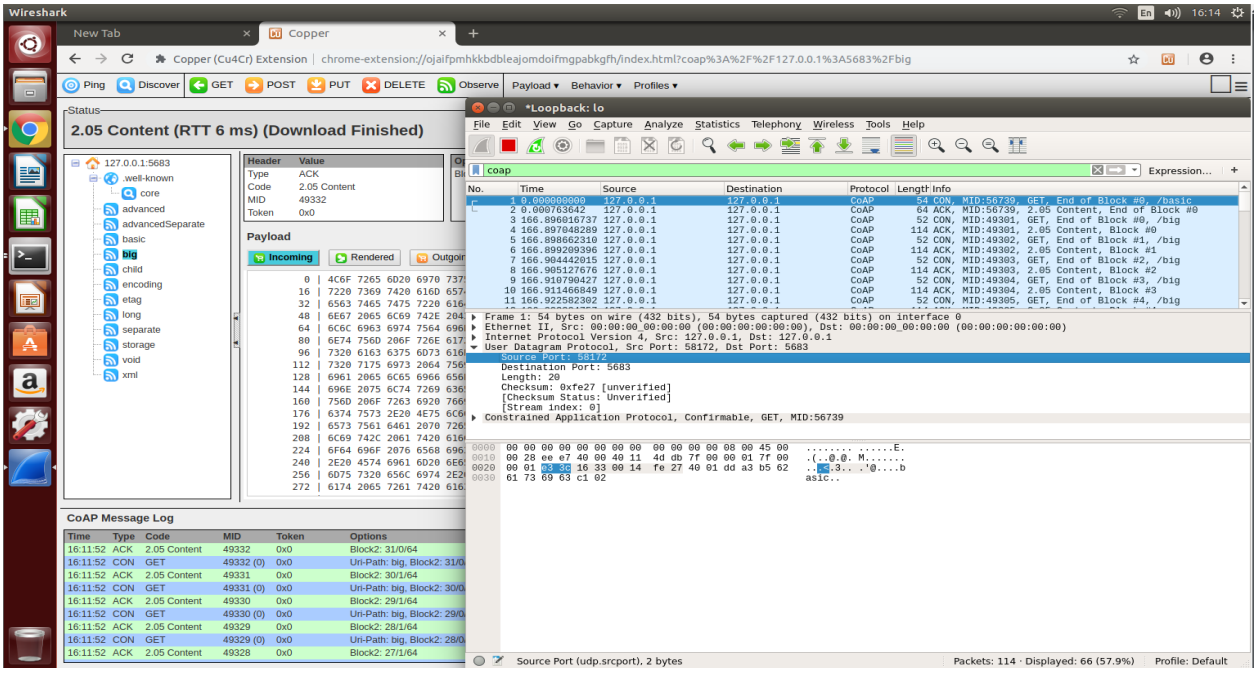


- **In the virtual machine, launch Wireshark ("sudo wireshark") and start a capture on the loopback interface. Repeat the last two actions in Chrome. Use COAP as a filter in Wireshark.**





- **From the capture identify the server and the client?**

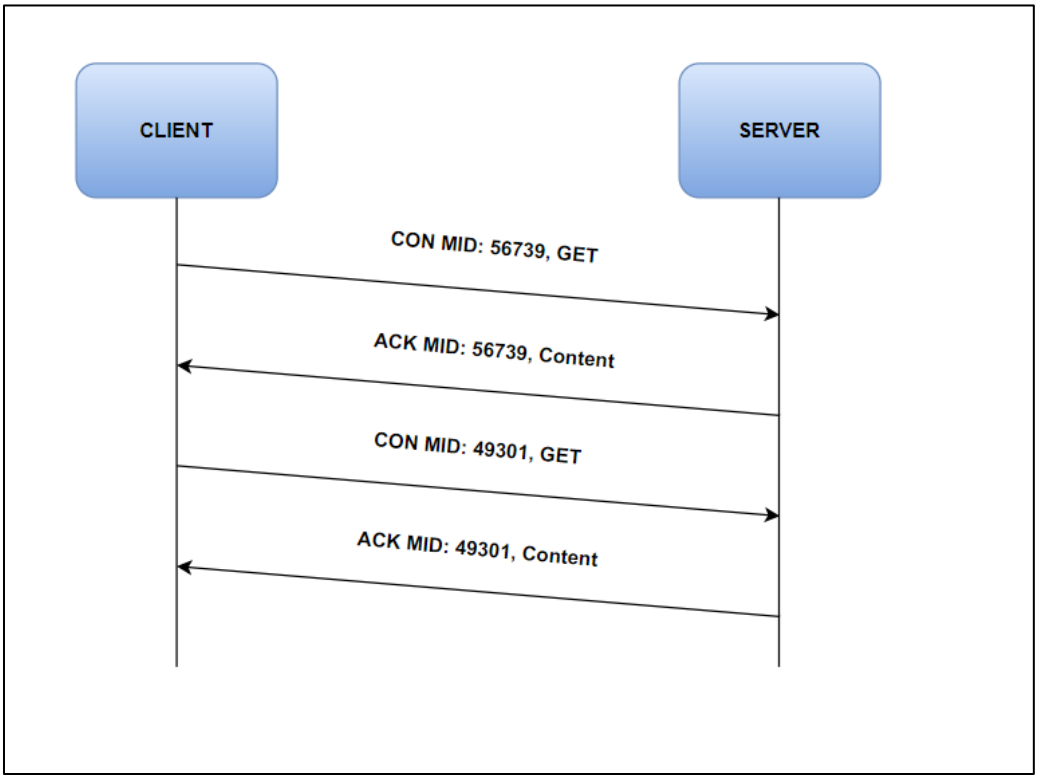


In the above captures, we can see that both the server and the client are running on the same local host 127.0.0.1, we may call this as a loopback, therefore the server and the client must be identified by the port numbers. All the COAP nodes are IPV6 multicast and listen to the default Coap port number.

Source Port: 58172

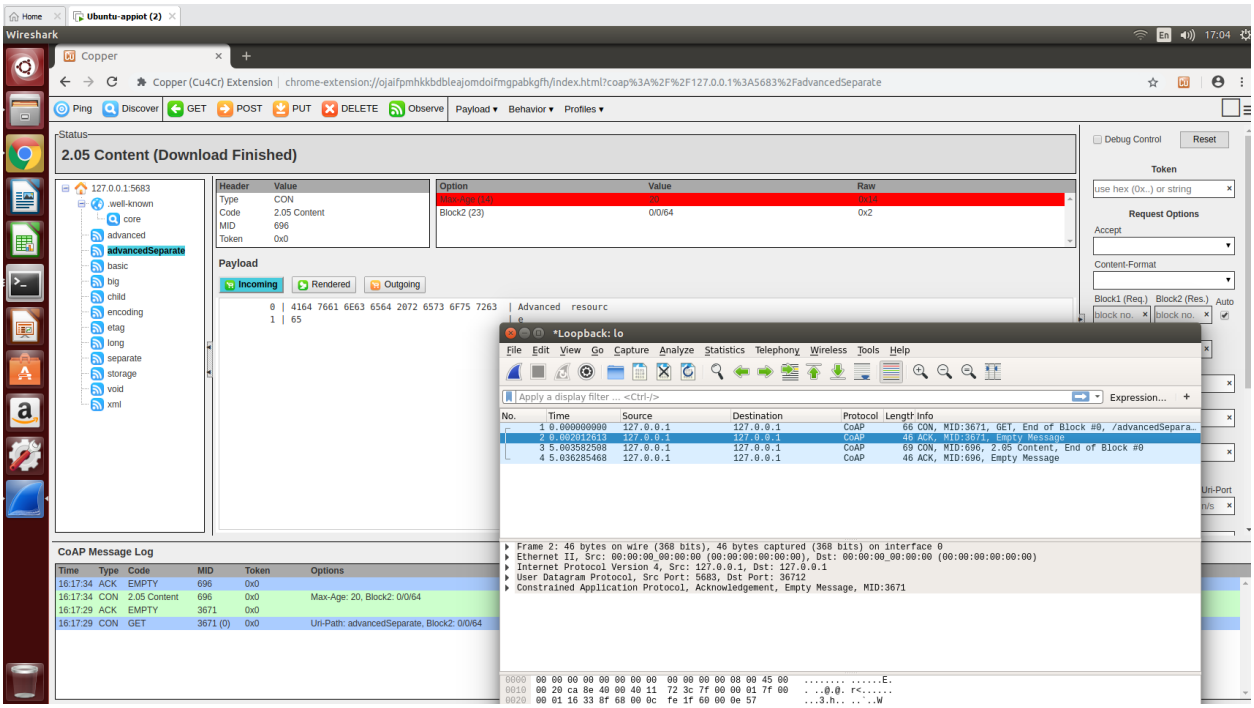
Destination Port: 5683

- **What are the differences between the two actions (i.e., big and basic)**
  Basic: It has minimum payload and so, the block size is minimum.
  Big: It has a maximum payload and so, it is fragmented to several blocks directly by the Coap protocol.
- **Draw the exchange for the second action and explain each step?**

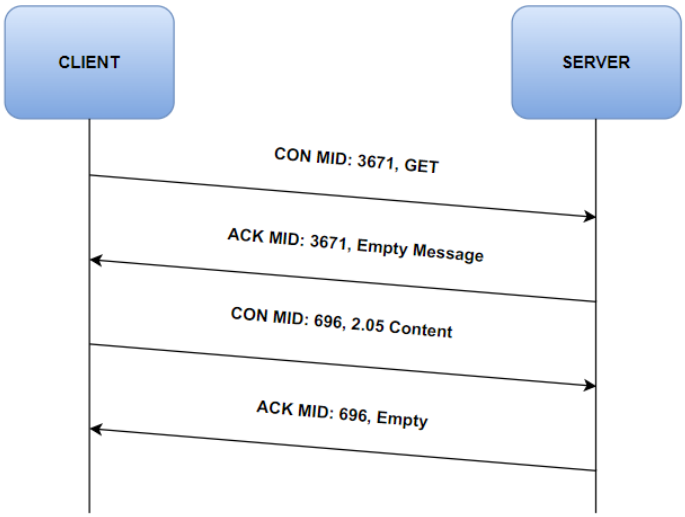


From the above exchange diagram, the client initially sends the GET request to the server in the confirmable message type with identifier 56739, then the server responds back with the ACK message to indicate the messages were reliable. Now, the GET request sent by the client to transmit the remaining block data in different message identifier 49301. In addition, for every CON message the server responds back with the same identifier number in the ACK message.

- **Start again a capture with Wireshark. In Chrome, select "advancedSeparate" and click on GET. Wait 5 seconds. Go to Wireshark and stop the capture.**



- **Draw the exchange and explain each step?**



The advancedSeparate is about the time delay resource, it has been delayed for about 5s to get the response from the server. From the capture, we can see that at Time 0.000000000 the client sends the request to the server, and immediately the server replies back to the client to indicate the reliable delivery separate mode. After 5s (Time at 5.003582508) the client again sends the request to the server to get a response. For every CON message the server responds back with the same message identifier number in ACK message.

- **Add a resource called "temp" to the server. The resource should send back a random value between 20 and 30, each time it is called. The data**

should be encoded using XML. The resource should be discovered and encoded as seen in the lecture on COAP.

- **Modify the code to delay the response of the server for 5 seconds, simulating a Readout action.**

```
class temp(Resource):
    def __init__(self, name="temp", coap_server=None):
        super(temp,self).__init__(name, coap_server, visible=True, observable=True, allow_children=True)
        self.value = random.randrange(20,30,1)
        self.payload = (defines.Content_types["application/xml"],"<value>" + str(self.value) + "</value>")
    def render_GET(self, request):
        time.sleep(5)
        self.value = random.randrange(20,30,1)
        self.payload = (defines.Content_types["application/xml"],"<value>" + str(self.value) + "</value>")
        return self
```

The above code defines a class called “Temp” which has two methods: __init__ and render_GET. The __init__ method initializes the instance of Temp with a name and a coap_server object which is set to None. It also sets some attributes such as visible, observable, and allow_children to Boolean value True. It then generates a random integer between 20 and 30, and sets the payload attribute to a tuple which contains the content type and a string representation of the value attribute.

The render_GET method then simulates a delay in response time by waiting for 5 seconds using the time.sleep() function. It then generates a new random integer between 20 and 30, updates the value attribute, and also updates the payload attribute to reflect the new value. Finally, it returns the updated resource using the self keyword.

```
#!/usr/bin/env python

import getopt
import sys
from coapthon.server.coap import CoAP
from exampleresources import BasicResource, Long, Separate, Storage, Big, voidResource, XMLResource, ETAGResource, \
    Child, \
    MultipleEncodingResource, AdvancedResource, AdvancedResourceSeparate, temp

__author__ = 'Giacomo Tanganelli'


class CoAPServer(CoAP):
    def __init__(self, host, port, multicast=False):
        CoAP.__init__(self, (host, port), multicast)
        self.add_resource('temp/', temp())
        self.add_resource('basic/', BasicResource())
        self.add_resource('storage/', Storage())
        self.add_resource('separate/', Separate())
        self.add_resource('long/', Long())
        self.add_resource('big/', Big())
        self.add_resource('void/', voidResource())
        self.add_resource('xml/', XMLResource())
        self.add_resource('encoding/', MultipleEncodingResource())
        self.add_resource('etag/', ETAGResource())
        self.add_resource('child/', Child())
        self.add_resource('advanced/', AdvancedResource())
        self.add_resource('advancedSeparate/', AdvancedResourceSeparate())

        print "CoAP Server start on " + host + ":" + str(port)
        print self.root.dump()


def usage():  # pragma: no cover
    print "coapserver.py -i <ip address> -p <port>"


def main(argv):  # pragma: no cover
    ip = "0.0.0.0"
    port = 5683
```

Now, in the Python code coap_server.py, a new resource name has been declared as “temp”. All the available resources can be discovered by the client in the copper extension by looking into all the Coap ports.

The random values generated are 22 and 23. There is a temp resource added and can be viewed on the left window.