UNIVERSITEIT VAN PRETORIA
UNIVERSITY OF PRETORIA
YUNIBESITHI YA PRETORIA

NavUP
Longsword Testing Report

Compiled By

Lucian Sargeant - u15225560
Ritesh Doolabh - u15075754
Peter Boxall - u14056136
Claude Greeff - u13153740
Harris Leshaba - u15312144
Hristian Vitrychenko - u15006442

GitHub Repository: COS 301 Team Longsword Data GitHub Repository(Phase 4)

2017

TEAM LONGSWORD (DATA)

Contents

1 Introduction

2 Introduction

For this phase we will be testing the Data module of the BroadSword Team. We have split the testing phase according to Functional Requirements, Non-Functional Requirements and Use Cases.

Their code was primarily coded in Python and used a NSQ message processing system. We will be testing the various cases and giving a brief description of how we tested followed by an explanation of the mark that was given to them.

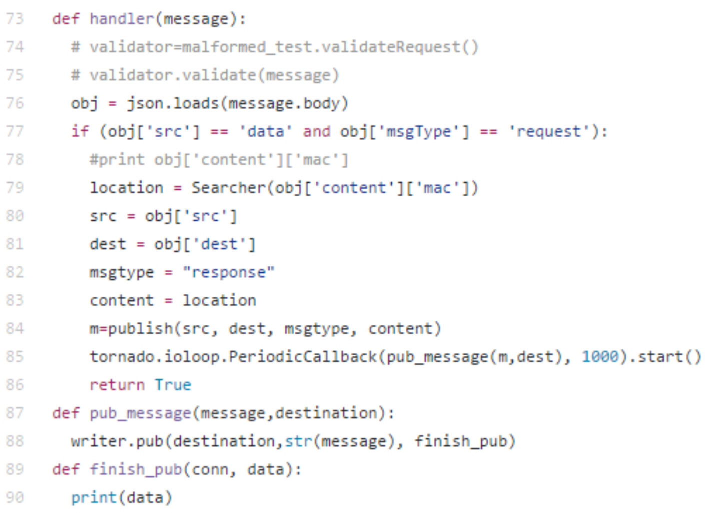
3 Service Contracts

3.1 Retrieving and passing device MAC address.

MARK: 10

The access module sends a location request to the data module via the NSQ server,the access module has to subscribes to a topic.The main entry point of the system, query resolver.py, sends the mac address to the aruba server in a query to get the location of the device. In query resolver.py, the handler function receives a request, it then passes the mac address to the searcher function-which will create a query to get the location for the device.

This functional requirement was fulfilled by the Broadsword team.



```python
def handler(message):
  # validator=malformed_test.validateRequest()
  # validator.validate(message)
  obj = json.loads(message.body)
  if (obj['src'] == 'data' and obj['msgType'] == 'request'):
    #print obj['content']['mac']
    location = Searcher(obj['content']['mac'])
    src = obj['src']
    dest = obj['dest']
    msgtype = "response"
    content = location
    m=publish(src, dest, msgtype, content)
    tornado.ioloop.PeriodicCallback(pub_message(m,dest), 1000).start()
    return True
def pub_message(message,destination):
  writer.pub(destination,str(message), finish_pub)
def finish_pub(conn, data):
  print(data)
```

Figure 1: Handler function.

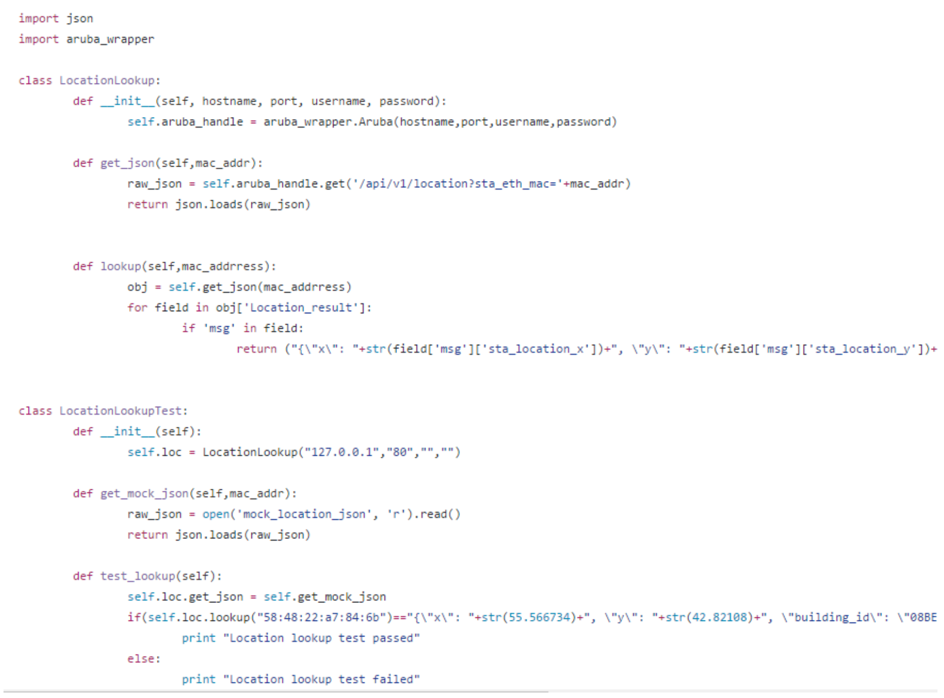
3.2 Logging in and maintaining a session with Aruba ALE.

3.3 Processing the request and retrieving location.

MARK: 10

In the python file query_resolver.py, Broadsword provides the options of using mock data to test their program or to follow standard procedure and use real data from Aruba by connecting to it through location_lookup.py, building_lookup.py and floor_lookup.py. The previously mentioned python files each connect to Aruba and all seperately log in by utilising aruba_wrapper.py which establishes and maintains a session with Aruba. Each class also filters their own JSON objects that are returned by Aruba and return only the necessary data thus fulfilling the requirement of processing requests and retrieving location.

This functional requirement was not fully fulfilled by the Broadsword team.



```
import json
import aruba_wrapper

class LocationLookup:
        def __init__(self, hostname, port, username, password):
                self.aruba_handle = aruba_wrapper.Aruba(hostname,port,username,password)

        def get_json(self,mac_addr):
                raw_json = self.aruba_handle.get('/api/v1/location?sta_eth_mac='+mac_addr)
                return json.loads(raw_json)


        def lookup(self,mac_addrress):
                obj = self.get_json(mac_addrress)
                for field in obj['Location_result']:
                        if 'msg' in field:
                                return ("{\"x\": "+str(field['msg']['sta_location_x'])+", \"y\": "+str(field['msg']['sta_location_y'])+


class LocationLookupTest:
        def __init__(self):
                self.loc = LocationLookup("127.0.0.1","80","","")

        def get_mock_json(self,mac_addr):
                raw_json = open('mock_location_json', 'r').read()
                return json.loads(raw_json)

        def test_lookup(self):
                self.loc.get_json = self.get_mock_json
                if(self.loc.lookup("58:48:22:a7:84:6b")=="{\"x\": "+str(55.566734)+", \"y\": "+str(42.82108)+", \"building_id\": \"08BE
                        print "Location lookup test passed"
                else:
                        print "Location lookup test failed"
```

Figure 2: aruba_wrapper.py

3.4 Returning a location to the source of the request.

MARK: 8

After the Aruba ALE connection is made by the aruba-wrapper class, the results of the location query are then delt with in the LocationLookup class. This class instantiates and exectues the aruba-wrapper implementation and then processes the JSON results to filter out what is needed.

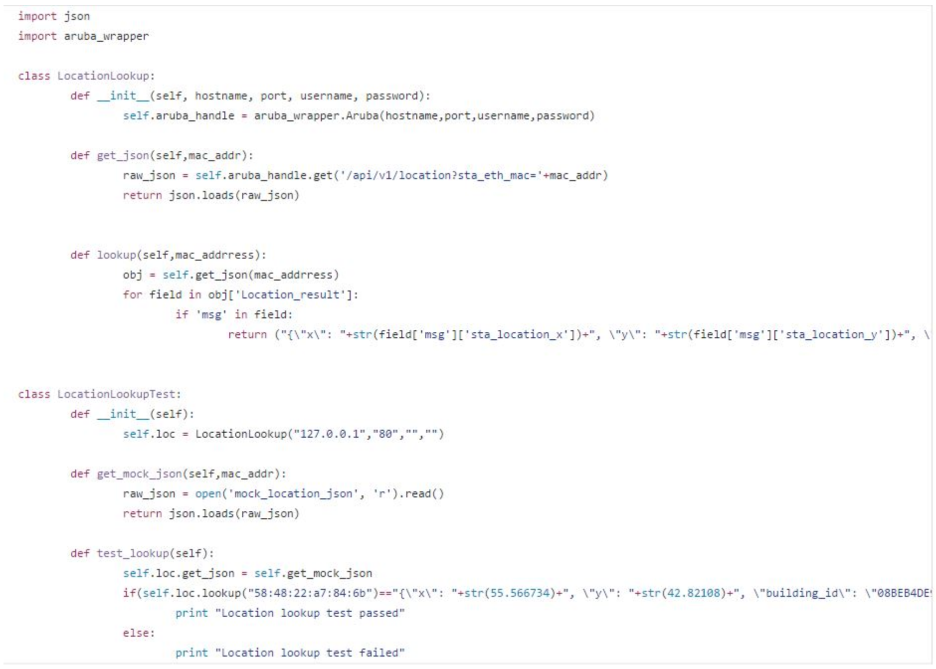
This functional requirement was fulfilled by the Broadsword team.

4 Non-Functional Requirements

4.1 Level of concurrency of the task.

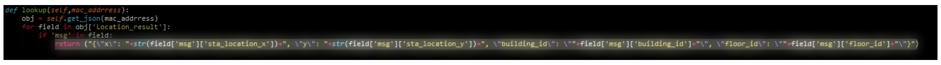
MARK: 0

Broadsword made use of a server known as NSQ which is widely known for its message passing based procedures. Whilst the NSQ core was built in Go (a programming language with built in concurreny), Broadsword did not seem to make use of any of those concurrent features in their code. This was made apparent in testNavigationConsumer.py where each NSQ action was made on after the other with no concurrency apparent anywhere in terms of mac address processing and message passing. In terms of concurrent programming in general, there is a none in any of the code presented by Broadsword.



```
import json
import aruba_wrapper

class LocationLookup:
        def __init__(self, hostname, port, username, password):
                self.aruba_handle = aruba_wrapper.Aruba(hostname,port,username,password)

        def get_json(self,mac_addr):
                raw_json = self.aruba_handle.get('/api/v1/location?sta_eth_mac='+mac_addr)
                return json.loads(raw_json)


        def lookup(self,mac_addrress):
                obj = self.get_json(mac_addrress)
                for field in obj['Location_result']:
                        if 'msg' in field:
                                return ("{\"x\": "+str(field['msg']['sta_location_x'])+", \"y\": "+str(field['msg']['sta_location_y'])+", \


class LocationLookupTest:
        def __init__(self):
                self.loc = LocationLookup("127.0.0.1","80","","")

        def get_mock_json(self,mac_addr):
                raw_json = open('mock_location_json', 'r').read()
                return json.loads(raw_json)

        def test_lookup(self):
                self.loc.get_json = self.get_mock_json
                if(self.loc.lookup("58:48:22:a7:84:6b")=="{\"x\": "+str(55.566734)+", \"y\": "+str(42.82108)+", \"building_id\": \"08BEB4DE
                        print "Location lookup test passed"
                else:
                        print "Location lookup test failed"
```

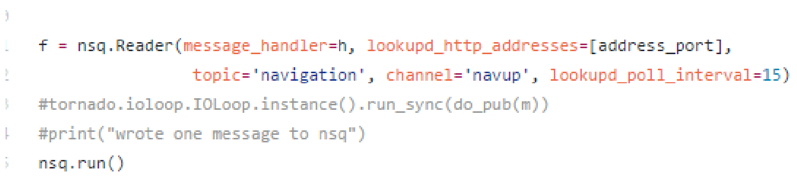
Figure 3: Mock JSON



```
def lookup(self,mac_addrress):
    obj = self.get_json(mac_addrress)
    for field in obj['Location_result']:
        if 'msg' in field:
            return ("{\"x\": "+str(field['msg']['sta_location_x'])+", \"y\": "+str(field['msg']['sta_location_y'])+", \"building_id\": \""+field['msg']['building_id']+"\", \"floor_id\": \""+field['msg']['floor_id']+"\"}")
```

Figure 4: JSON returned

This non-functional requirement was not fulfilled by the Broadsword team.



```
    f = nsq.Reader(message_handler=h, lookupd_http_addresses=[address_port],
                   topic='navigation', channel='navup', lookupd_poll_interval=15)
    #tornado.ioloop.IOLoop.instance().run_sync(do_pub(m))
    #print("wrote one message to nsq")
    nsq.run()
```

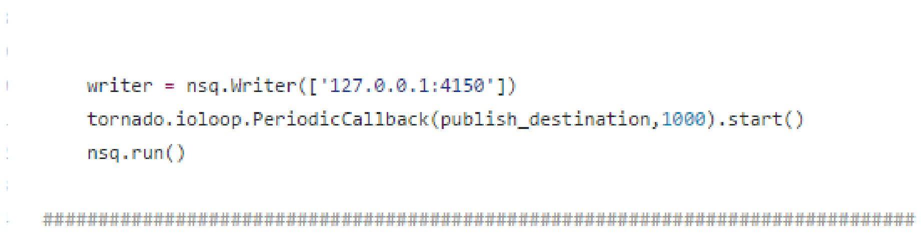
Figure 5: NSQ Use Without Concurrency

4.2 Performance of the request processing.

MARK: 1

Broadsword used a NSQ server which is a message passing system that primarily relies on queue structures to store information. Although NSQ does allow for concurrent programmability, these features were not used efficiently enough to allow large numbers of requests to be "streamed" at high speeds. We simply used a timer in their code to count the number of instances of the location object that was returned for a specific MAC address. This terminated the code after 60 seconds and gave the final number of instances returned.

As can be seen from the screen-shots below, within the duration of 60 seconds, the system only managed to return 63 Location objects for a single mac address. In terms of real-world functionality, this would mean a significant bottleneck on the system. This is because the system

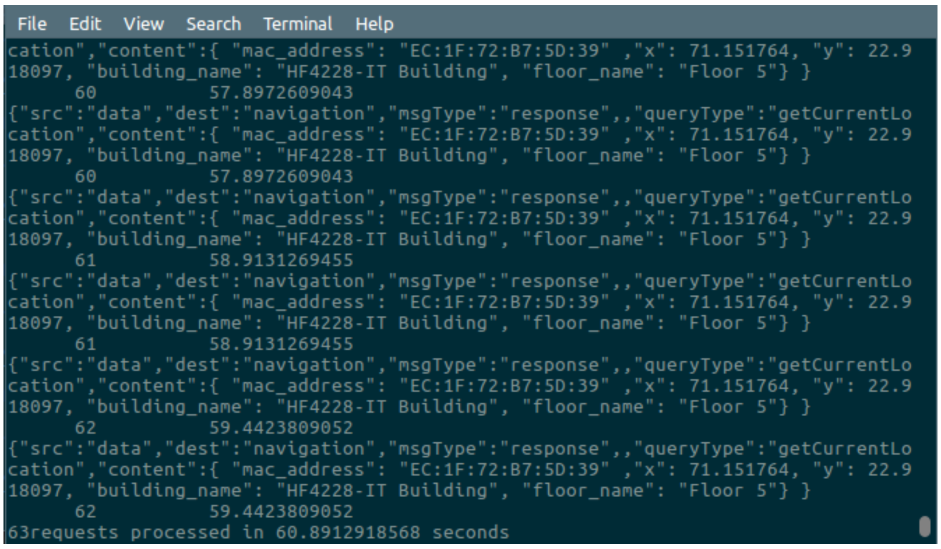


```
    writer = nsq.Writer(['127.0.0.1:4150'])
    tornado.ioloop.PeriodicCallback(publish_destination,1000).start()
    nsq.run()

##################################################################################
```

Figure 6: NSQ Writer

would need at a minimum roughly 30 000 requests to be fulfilled in a matter of 60 seconds. In the regard of high speed data streaming, the Broadsword Data module has failed.

This non-functional requirement was not fulfilled by the Broadsword team.



```
cation","content":{ "mac_address": "EC:1F:72:B7:5D:39" ,"x": 71.151764, "y": 22.9
18097, "building_name": "HF4228-IT Building", "floor_name": "Floor 5"} }
      60          57.8972609043
{"src":"data","dest":"navigation","msgType":"response",,"queryType":"getCurrentLo
cation","content":{ "mac_address": "EC:1F:72:B7:5D:39" ,"x": 71.151764, "y": 22.9
18097, "building_name": "HF4228-IT Building", "floor_name": "Floor 5"} }
      60          57.8972609043
{"src":"data","dest":"navigation","msgType":"response",,"queryType":"getCurrentLo
cation","content":{ "mac_address": "EC:1F:72:B7:5D:39" ,"x": 71.151764, "y": 22.9
18097, "building_name": "HF4228-IT Building", "floor_name": "Floor 5"} }
      61          58.9131269455
{"src":"data","dest":"navigation","msgType":"response",,"queryType":"getCurrentLo
cation","content":{ "mac_address": "EC:1F:72:B7:5D:39" ,"x": 71.151764, "y": 22.9
18097, "building_name": "HF4228-IT Building", "floor_name": "Floor 5"} }
      61          58.9131269455
{"src":"data","dest":"navigation","msgType":"response",,"queryType":"getCurrentLo
cation","content":{ "mac_address": "EC:1F:72:B7:5D:39" ,"x": 71.151764, "y": 22.9
18097, "building_name": "HF4228-IT Building", "floor_name": "Floor 5"} }
      62          59.4423809052
{"src":"data","dest":"navigation","msgType":"response",,"queryType":"getCurrentLo
cation","content":{ "mac_address": "EC:1F:72:B7:5D:39" ,"x": 71.151764, "y": 22.9
18097, "building_name": "HF4228-IT Building", "floor_name": "Floor 5"} }
      62          59.4423809052
63requests processed in 60.8912918568 seconds
```

Figure 7: Number of requests processed in 60 Seconds

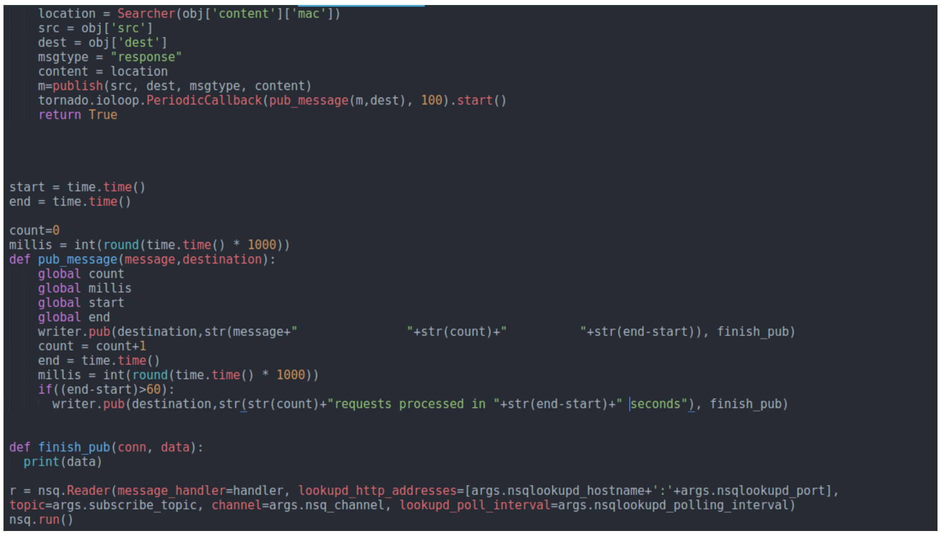
4.3 Maintainability and modularity of the code and repository.

4.4 Integrability and ease of transfer into a final system.

MARK: 5

Whilst the code is very modular in format allowing for certain pieces to be easily matched to other systems, the code appears to be fragmented to a stage where there are a lot of inter-dependency between these classes. Meaning It would be harder to break the system up and mould it to a new system. Also the lack of commenting makes it difficult for one to see where things are implemented and what certain aspects of the code does. This makes it harder for a developer who is required to integrate the code who may not have necessarilly written it.

This non-functional requirement was partially fulfilled by the Broadsword team.



```python
    location = Searcher(obj['content']['mac'])
    src = obj['src']
    dest = obj['dest']
    msgtype = "response"
    content = location
    m=publish(src, dest, msgtype, content)
    tornado.ioloop.PeriodicCallback(pub_message(m,dest), 100).start()
    return True



start = time.time()
end = time.time()

count=0
millis = int(round(time.time() * 1000))
def pub_message(message,destination):
    global count
    global millis
    global start
    global end
    writer.pub(destination,str(message+"               "+str(count)+"          "+str(end-start)), finish_pub)
    count = count+1
    end = time.time()
    millis = int(round(time.time() * 1000))
    if((end-start)>60):
      writer.pub(destination,str(str(count)+"requests processed in "+str(end-start)+" seconds"), finish_pub)


def finish_pub(conn, data):
  print(data)

r = nsq.Reader(message_handler=handler, lookupd_http_addresses=[args.nsqlookupd_hostname+':'+args.nsqlookupd_port],
topic=args.subscribe_topic, channel=args.nsq_channel, lookupd_poll_interval=args.nsqlookupd_polling_interval)
nsq.run()
```

Figure 8: Code inserted to test the number of requests processed in 60 seconds

5 Use Cases

Broadsword has made use of a NSQ. A 'nsqd' instance is designed to handle multiple streams of data at once. In NSQ terms, streams are called "topics" where a topic can have 1 or more "channels". A channel maps to a downstream service consuming a topic. The method broadsword used was to create a topic, "data", through subscribing to channel "navup", which is a channel on the "data" topic. The topic is created through the first subscription. This method lets channels and topics buffer their data independently. This allows for fast downstream, preventing a slow consumer to cause a bottleneck or delay. Successful upstream and downstream

communication ticks the use case boxes, although more efficient changes could be made for concurrent processing, a mark of 8 out of 10 for downstream and upstream seems fair.

5.1 Upstream communication.

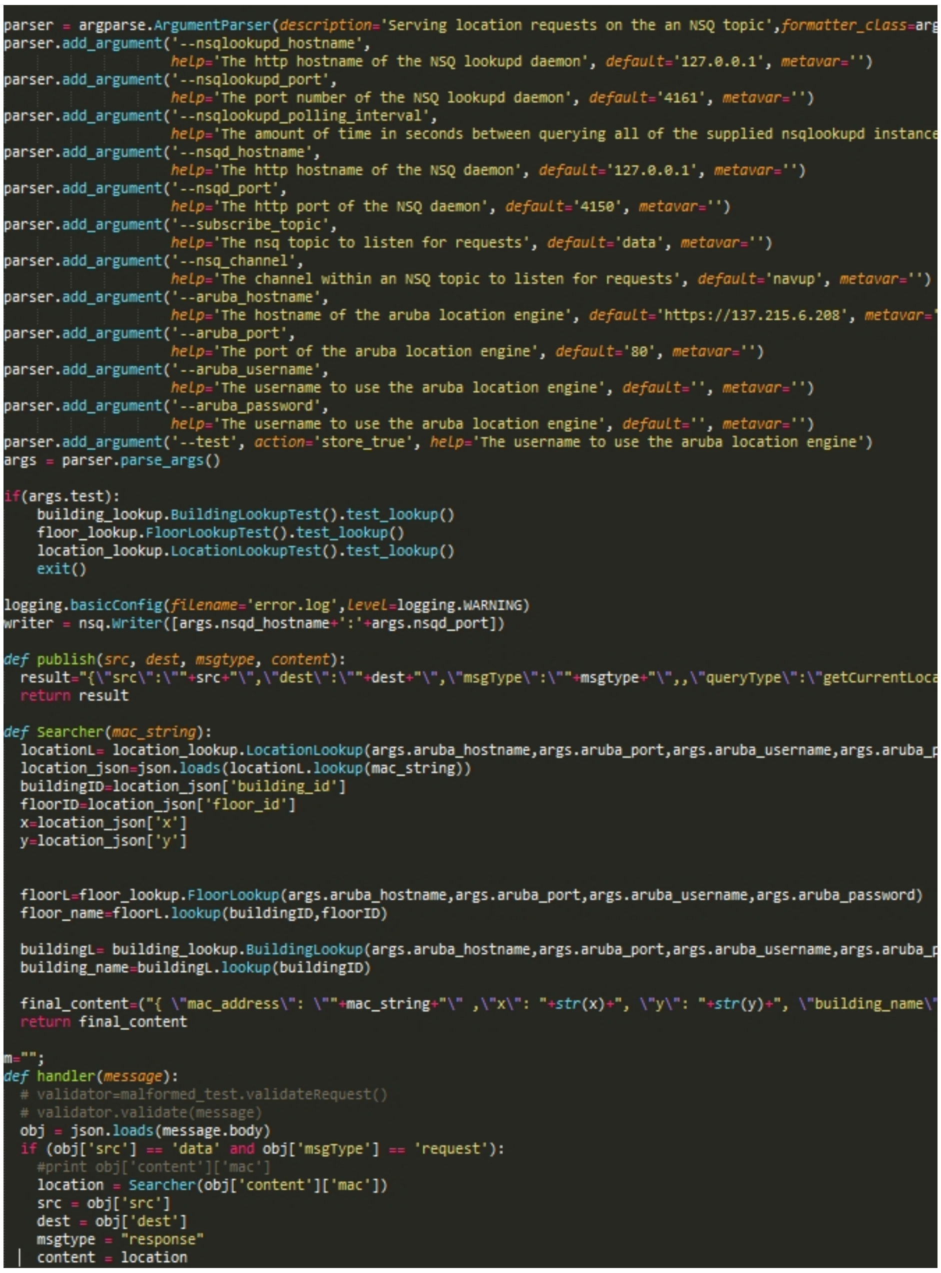
MARK: 8

The file named "publisher.py" will write to nsqd port 4150, this is their upstream communication channel.

5.2 Downstream communication.

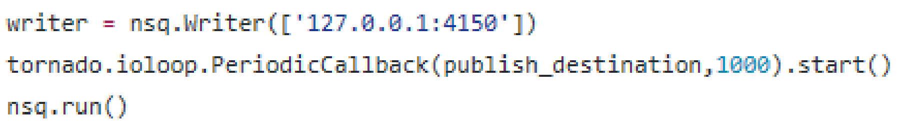
MARK: 8

Consumers make use of a HTTP /lookup endpoint. Consumers are introduced to topics through making use of the addresses of Broadsword's nsqlookupd instance. In their case it would be host name '127.0.0.1' and port '4161'. This would be their downstream communication.

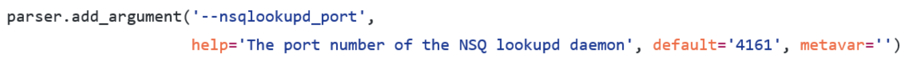


```
parser = argparse.ArgumentParser(description='Serving location requests on the an NSQ topic',formatter_class=arg
parser.add_argument('--nsqlookupd_hostname',
                    help='The http hostname of the NSQ lookupd daemon', default='127.0.0.1', metavar='')
parser.add_argument('--nsqlookupd_port',
                    help='The port number of the NSQ lookupd daemon', default='4161', metavar='')
parser.add_argument('--nsqlookupd_polling_interval',
                    help='The amount of time in seconds between querying all of the supplied nsqlookupd instance
parser.add_argument('--nsqd_hostname',
                    help='The http hostname of the NSQ daemon', default='127.0.0.1', metavar='')
parser.add_argument('--nsqd_port',
                    help='The http port of the NSQ daemon', default='4150', metavar='')
parser.add_argument('--subscribe_topic',
                    help='The nsq topic to listen for requests', default='data', metavar='')
parser.add_argument('--nsq_channel',
                    help='The channel within an NSQ topic to listen for requests', default='navup', metavar='')
parser.add_argument('--aruba_hostname',
                    help='The hostname of the aruba location engine', default='https://137.215.6.208', metavar='
parser.add_argument('--aruba_port',
                    help='The port of the aruba location engine', default='80', metavar='')
parser.add_argument('--aruba_username',
                    help='The username to use the aruba location engine', default='', metavar='')
parser.add_argument('--aruba_password',
                    help='The username to use the aruba location engine', default='', metavar='')
parser.add_argument('--test', action='store_true', help='The username to use the aruba location engine')
args = parser.parse_args()

if(args.test):
    building_lookup.BuildingLookupTest().test_lookup()
    floor_lookup.FloorLookupTest().test_lookup()
    location_lookup.LocationLookupTest().test_lookup()
    exit()

logging.basicConfig(filename='error.log',level=logging.WARNING)
writer = nsq.Writer([args.nsqd_hostname+':'+args.nsqd_port])

def publish(src, dest, msgtype, content):
  result="{\"src\":\""+src+"\",\"dest\":\""+dest+"\",\"msgType\":\""+msgtype+"\",,\"queryType\":\"getCurrentLoca
  return result

def Searcher(mac_string):
  locationL= location_lookup.LocationLookup(args.aruba_hostname,args.aruba_port,args.aruba_username,args.aruba_p
  location_json=json.loads(locationL.lookup(mac_string))
  buildingID=location_json['building_id']
  floorID=location_json['floor_id']
  x=location_json['x']
  y=location_json['y']


  floorL=floor_lookup.FloorLookup(args.aruba_hostname,args.aruba_port,args.aruba_username,args.aruba_password)
  floor_name=floorL.lookup(buildingID,floorID)

  buildingL= building_lookup.BuildingLookup(args.aruba_hostname,args.aruba_port,args.aruba_username,args.aruba_p
  building_name=buildingL.lookup(buildingID)

  final_content=("{ \"mac_address\": \""+mac_string+"\" ,\"x\": "+str(x)+", \"y\": "+str(y)+", \"building_name\"
  return final_content

m="";
def handler(message):
  # validator=malformed_test.validateRequest()
  # validator.validate(message)
  obj = json.loads(message.body)
  if (obj['src'] == 'data' and obj['msgType'] == 'request'):
    #print obj['content']['mac']
    location = Searcher(obj['content']['mac'])
    src = obj['src']
    dest = obj['dest']
    msgtype = "response"
    content = location
```

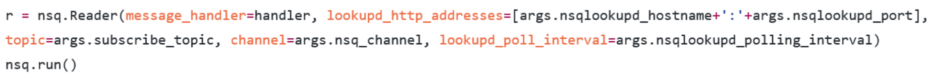
Figure 9: No Comments



```
writer = nsq.Writer(['127.0.0.1:4150'])
tornado.ioloop.PeriodicCallback(publish_destination,1000).start()
nsq.run()
```



```
parser.add_argument('--nsqlookupd_port',
                    help='The port number of the NSQ lookupd daemon', default='4161', metavar='')
```



```
r = nsq.Reader(message_handler=handler, lookupd_http_addresses=[args.nsqlookupd_hostname+':'+args.nsqlookupd_port],
topic=args.subscribe_topic, channel=args.nsq_channel, lookupd_poll_interval=args.nsqlookupd_polling_interval)
nsq.run()
```