COP505 Coursework Report

By Matthew, Nathan, Simon and Yiwen

Table of Contents

1 - Abstract

The project that this report focuses on was a task to develop a mimic of a base station system with dynamically generated mobile phone connections. The report will look at the positives and negatives of the development, the details of the design implemented, and potential future improvements that could be made, and how the project met the required tasks set. It will also a feature a full set of tests to make sure the system functions as required.

2 - Instructions of use

2a - Pre-requisite conditions

The first condition is that the system must be run on the university computers. The system only works in the Linux or OSX partition.

The user must install “pymongo” and “requests” onto their system using the following terminal commands:

```
pip2 install requests --user
pip2 install pymongo –-user
```

Firstly, put the files under the Home (~) directory.

The user must edit this line in the files basestation.py and monitor.py:

```
url = "http://cosyly@sci-project.lboro.ac.uk”
```

The user should replace the bold word with their own username for the sci-project. This should be found on line 62, 94 of basestation.py and line 4 of monitor.py

In the dataprocess.py file, the user must edit line 9 and 10:

```
client = MongoClient("mongodb://cosyly:4JbHBLrBqK@sci-project:27017/cosyly")
db = client.cosyly
```

The user should replace the bold words with their own username and password for the mongo database.

Next, open a new terminal to connect to the sci-project server. To do this, enter:

```
ssh <username>@sci-project.lboro.ac.uk
```

On your local terminal, enter the following command:

```
scp dataprocess.py <username>@<username>.sci-
project.lboro.ac.uk:dataprocess.py
```

This should upload the dataprocess.py to the server. You can check this by using the ‘ls’ command in the server terminal.

Now, when running dataprocess.py on the sci-project server, the user must do so by running the following in the server terminal:

```
export FLASK_APP=dataprocess.py
flask run --host=0.0.0.0 --port=5001
```

When running the base station, and network monitoring files, the user must compile using the command ‘python2’ rather than simply ‘python’ on a local terminal, for example:

```
python2 basestation.py
```

3 - Details of System Design

3a - Base Station

The base station utilises multithreading at its core. One for mobile phone connecting, mobile phone disconnecting, one for error catching and the main thread to manage these threads. The As they run concurrently, mobile phones are able to connect to a base station (be created and inserted to the database) and disconnect from the base station (be deleted from the database) at the same time, which means that occasionally, their timestamps will be the same. This offers a level of complexity to the project that we might not have needed to include, but felt that it was something that we could include in time.

3b - Data Processor

We opted to use Python Flask for our project because it's incredibly lightweight and perfect for a project of the 'small' nature such as this. It was incredibly easy to learn how to use in such a short space of time, and allowed the base to make the project as we wanted to, meeting the project specification. Largely, the major functions of this file are there to store data in the database, remove data from the database and also to pull data from the database and then display it to the user through another function in the system.

Method	**URI**	**Description**
GET	/	Test connection to server, print and return 'This is fine' in the server terminal.
GET	/basestation	Lists all base stations and their data, including their capacity, time of start, location and ID
POST	/basestation	Used for creating a new base station. Takes data from user input to create the object
GET	/basestation/getid/<id>	Returns all mobile phones currently connected to a specific base station
GET	/basestation/getloc/<city>	Returns all base stations that have the same 'city' as one of their properties
DELETE	/basestation/<id>	Deletes the given base station (by ID) from the database
GET	/basestation/busy	Returns a list of base stations which are currently at 90% capacity or over
POST	/mobiledata	Inserts a 'mobile phone' into the database along

		with the base station ID where it was created
DELETE	/mobiledata/<id>/<number>	Deletes a mobile phone (disconnects) from the database, given the base station ID and mobile number
GET	/basestation/stat	Get all the capacity, the number of current connect to the base stations and the return the percentage of the total capacity used for each base station.

3c - Network Monitor

The network monitor developed for this project allows for all of required tasks in the specification, and a number of additional tasks that have been included, this includes getting all the mobile phone connections of a base station by the base station ID, retrieving all of the base stations within a single city (an attribute passed by the user into the database). The network monitor also has the ability to catch any network, HTTP or request errors, and if any are detected, the program will retry the connection 5 times at a separation of 3 seconds each (attempting to give adequate time for the connection to be re-established), and if reconnection is failed, then the error is returned and displayed to the user.

4 - Justifications and Decisions

Initially, our group's first effort to create the base station model was to separate the base station and mobile phone generation programs. This was more complex than was necessary as it turned out. Therefore, we have decided that there is no need for a TCP connection to simulate the mobile phone connections (meaning that the mobile phone generation can occur inside the base station file) but simply created (randomly generated).

With the TCP connection method, there will be a significantly large amount of TCP connections depending on the base station capacity which is not ideal when using the lab computers. In addition, we were told that this high-level implementation is not necessary for the coursework given the time span we were given to complete this coursework. This way, there does not need to be a TCP server on the server side. As a result, we included the mobile phone generation in the base station file. This made the project simpler as all the code needed on the client side was combined into a single place, which made development easier.

Another assumption made during development is that we are able to open the base station in multiple terminal windows (to simulate multiple instances base stations).

Before we made this assumption, the group attempted to simulate multiple base stations by using different sockets for each base station (meaning different ports for each instance). Unfortunately, this would not work very well as we will be utilising multiple ports in one file, which would be incredibly complicated and it could potentially represent a security risk, as the user may be tempted to open more ports to allow for more base stations, some of which may be vulnerable to outside attack.

With this assumption, the group had decided to not utilise multithreading for simulating each base station instance. This decision was made because the program might not then meet the project specification. This is because the method might only be simulating a single base station with multiple antennae rather than different, distinct base stations.

Our group settled on a decision to use multithreading to simulate mobile phone connections, disconnections and error handling. This way, multiple terminal windows could simulate the base stations, and on each terminal window, the base station would be generating its own mobile phone connections. This made the project significantly more scalable as we had a set, core group of three functions for multithreading and each one running independently of the other two, which allowed the project to meet the 'random' specifications.

As there are no limits on terminal windows, the user can open as many as they like, with each individual window effectively acting as a base station which satisfy the condition where multiple instances of base stations are expect to run on a local computer.

5 - Implementation of tasks

5a - Base Station

The first task specifies that the group must simulate a base station with three attributes: a unique ID, a location (including a city, county, and street name) and a base station capacity. *Figure 1(a)* shows how we managed to generate this data, with all information except the ID being inputted by the user, and then it being sent by an HTTP put request to the server, where it is later stored in the database. If the request is not successful, it will retry five times. If all fails, it will terminate the program.

It is specified that the base station must inform the rest of the system that it is 'starting up' when it is created, and that on this message, it should include the properties of the base station, and its timestamp of creation. This is shown in *Figure 1(b)*.

Thirdly for the base station, the system must simulate mobile phones connecting (we chose to do this locally, inside the base station file, rather than in its own file) and leaving it. When connecting, the mobile phone's data (message, number, time of connection and which base station it is connected to) must be recorded and stored in the database, and upon leaving the base station, this information must be deleted from the database. This is all present in our system, and is shown in *Figure 1(c)*.

The main thread will start three threads, those being one for connection, one for disconnection and one for error handling.

For the connection thread, it will continuously generate mobile connections which sends HTTP POST requests to the server and they are recorded in the database. After a phone connection is generated, this connection is put into a priority queue, the disconnection thread which is responsible for disconnection (HTTP DELETE request) gets the connection from the queue and sends a disconnection request to the server.

The main thread can control the generation of mobile connection and termination of the base station by using the command 'startpause' and 'stop'. Termination of the base station will remove the record of the base station in the database.

Another specification of the base station is that the data flow (mobile phone connections and disconnections) must occur at random intervals of time, and specifically not be hard-coded. To achieve this, we utilised multithreading, time.sleep() and random function. The multithreading allows the program to concurrently generating different phone connecting (sending HTTP POST requests) and disconnecting a phone connection (sending HTTP DELETE requests), ocassionally at the same moment in time. The time.sleep() and random function will provide random wait time before generate or delete a connection. This is shown in *Figure 1(d)*.

An additional feature included in this program is that we are able to limit the number of connections per base station specified by the user during the start-up of the base station. When the base station has reach its maximum connection limit, the thread will wait until a connection leaves the base station (a HTTP DELETE request is made) before generating a new phone connection.

Finally, the base station must have an offline functionality. This meant that if the HTTP request to the server fails, the data must be stored and the system must be prepared to send the data to the server at a later time, when the server connection is re-established. We have two methods to achieve this. If the program fails to send an HTTP POST request for the start-up of the base station, we set up a loop with a reasonable time restraint on time of re-connection, if it is reconnected in time, then the data is sent, otherwise it has 5 attempts at sending, and if all fail, the program will terminate. If the program fails to send an HTTP request related to the phone connection, the data is stored in a separate queue. When the server is resumed, the error handling thread which handles the error handling will get the data from the queue and make the HTTP request to the server. However, if the program closes when the server is not available, the data is lost as it would be uneconomical to store data indefinitely. This is shown in *Figure 1(e)*.

5b - Data Processor

The ‘data processor’ in comparison to the base station was quite small, with fewer, yet somewhat more complex tasks. Firstly, the data processor must have the ability to run continually. This was quite simple as we selected Flask to develop our API, it already had the functionality to run continually with a terminal command which must be executed in order for the system to work. This is specified above.

The data processor must provide a series of restful API’s, these are shown in *Section 3b* of the report, and are set out in such a way that base station data manipulation and mobile phone data manipulation are separated. They have the ability to communicate with the base station simulator using an HTTP response.

Finally, the data processor has the ability to store base station and mobile phone data (shown in *Figure 1(c)*) in the database, and remove the information as needed by the system requirements. Database storing and deletion is shown in *Figure 2*. The data processor specifies the two sets of data must be in two separate collections, as specified in the project specification.

5c - Network Monitor

The network monitor application needed the ability to query the database (using the data processor) and return it to the user. To achieve this, our group uses HTTP requests to get data from the data processor, which will then query the database and return the requested data to the network monitor for manipulation.

The main functions of this program are to be able to show a list of bases station activated and a list of the busiest base stations where their capacity if over 90%. The capability to list all of the base station data (meaning the base station ID, name, location and time of creation) created by the user, as shown in *Figure 3(a)* is included. Also, the capability to provide a list of base stations at 90% or more of their own capacity has been included. This is shown in *Figure 3(b).*

The other major require functionality of the project is to retrieve statistical analysis from the data. Our group decided to get the average number of mobile phone connections over the total number of base stations. This can be seen in *Figure 3(c).*

Other function in this program included getting the current list of mobile connection from a specific base station and getting a list of base stations from a specific location. These functionalities are relatively similar, given the way the program sends an HTTP GET request to the server and return the query data for the program to manipulate and display to the user. If the request fails to send, it will retry for five time. If all fail, then the program will terminate.

6 - Future Improvements

6a - Base Station

A realistic future improvement to the base station program would be that when the base station reaches a critical capacity threshold, a new base station could be created to relieve pressure on that former base station. This would be incredibly useful as it would allow for a large amount of scalability for the project. An alternative to this improvement would be that when the base station reaches critical capacity, the program could move a number of connections to another base station within a reasonable vicinity of the previous base station.

The potential for future improvements on the current base station file is limited at best under the specification, but looking outside of the specification, it is quite vast. One of the improvements that could be made would be on the random generation of mobile phones and the phones disconnecting from the base station, which could be more organic. By this, we mean that the time between the two is not simply a random number, but based on a complex algorithm which could move mobile phones between base stations, effectively mimicking reality.

Along with that, the base station could pull location data from the internet and allow the user to choose from a wide variety of real locations, rather than the user simply inputting a string as their location data. This would ensure that all location information is real, meaning the program would be more applicable to the potential real-world application, rather than simply a project.

6b - Network Monitor

For the network monitor, we could potentially allow the database to store more data regarding timestamps of the mobile phone connections and base station initialisations, which would allow us to plot graphs with information like average connections on a specific base station, or over the entire collection of base stations. This would be much better feedback for the user than simply a text output in the terminal.

A more ambitious future improvement for the network monitor could have a proper graphical user interface for the network monitor. This would be better for the user as it's significantly more user-friendly, and easier to understand (making it easier to learn about). This would also increase the scope for the project as we would not be limited by a numerical options list, but could have a statistical analysis search function.

7 - Testing

7a - Test cases

ID	Description	Steps	Expected outcome	Pass/Fail
1	Start server	Run ‘flask run –host=0.0.0.0 –port=5001’	Server to start on the localhost and at the port specified	Pass
2	Start basestationfin.py	Run ‘python2 basestationfin.py’	User to be asked for username, name, location and capacity of the base station	Pass
3	Start monitorfin.py	Run ‘python2 monitorfin.py’	User to be presented with a list of options available for retrieving information	Pass
4	Generate base station	Follow instructions on the terminal after running ‘basestationfin.py’	An HTTP put request is sent to the server including the information given buy the user following the on-screen instructions	Pass
5	Server inserts base station data into database	Send data from ‘basestationfin.py’	When server receives HTTP request from the user, it will insert the base station data into the database	Pass
6	Generate mobile phone connections	User will be prompted on-screen to begin generating connections, they must follow the prompt	Mobile phone connections are generated at random intervals and sent to the server	Pass
7	Server inserts mobile phone data into database	Send data from ‘basestationfin.py’	When server receives HTTP request from the base station file, it will insert the mobile phone data into the database	Pass
8	Stop mobile phone creation	After creating a base station instance, insert ‘stop’ into the terminal	The base station should cease creation of the mobile phones and begin deleting the mobile phones using a DELETE HTTP request	Pass
9	Pause mobile phone creation	After creating a base station instance, insert ‘startpause’ into the terminal	The base station should cease creation of the mobile, but not delete them	Pass

10	Unpause mobile phone creation	After creating a base station instance, insert ‘startpause’ into the terminal	The base station should once again begin creation of the mobile phones	Pass
11	Display a list of base stations	After starting ‘monitorfin.py’, the user must insert ‘1’ into the terminal when prompted	The terminal should output a list of base stations (if there are any) and the relevant base station information in a neat format	Pass
12	Display a list of base stations at 90% or more capacity	After starting ‘monitorfin.py’, the user must insert ‘2’ into the terminal when prompted	The terminal should output a list of base stations (if there are any) and the relevant base station information in a neat format	Pass
13	Display a list of mobile phone connections to a single base station	After starting ‘monitorfin.py’, the user must insert ‘3’ into the terminal when prompted, they must then insert the base station alphanumerical ID	The terminal should output a list of mobile phone connections to that specific base station, in a neat format	Pass
14	Display a list of base stations grouped by their ‘city’ attribute	After starting ‘monitorfin.py’, the user must insert ‘4’ into the terminal when prompted, they must then insert a ‘city’	The terminal should output a list of base stations that are connected to that ‘city’ in a neat format	Pass
15	Display the average number of connections across all base stations	After starting ‘monitorfin.py’, the user must insert ‘5’ into the terminal when prompted	The terminal should output a list of base stations and then also display the average number of connections	Pass
16	Terminate ‘monitorfin.py’	After starting ‘monitorfin.py’, the user must insert ‘7’ into the terminal when prompted	The program will cease running in the terminal	Pass

8 - Appendices

8a - Figure 1

a) *User specifying the base station data*

```
stationName = raw_input("\nName of the station: ")
stationLocation = raw_input("City of the station: ")
streetname = raw_input("Street name of the station: ")
statcounty = raw_input("County of the station: ")
while True:
    try:
        stationCapacity = int(input("Capacity of the station: "))
        break
    except:
        print 'Enter a number'



global_count = stationCapacity

station = basestation(stationName, stationLocation,streetname,statcounty, stationCapacity)
data = {'id':station.id, 'name':station.name, 'street':streetname,'county':statcounty, 'location':station.location, 'capacity':station.capacity, 'time':time.time()}
```

b) *The server notifying that a base station has been created*

```
@app.route('/basestation', methods=['GET', 'POST'])
def insert_stations():
    if request.method == 'POST':
        data1 = json.loads(request.data)
        db.BaseStation.insert_one({"stationid":data1['id'],
                "stationname":data1['name'],
                "location":data1['location'],
                "capacity":data1['capacity'],
                'timestarted':data1['time'],
                'street':data1['street'],
                'county':data1['county']
        })
        print 'id: '+str(data1['id'])+'\nBasestation: '+data1['name']+'\nLocation: '+data1['location']+'\nCapacity: '+str(data1['capacity'])+'\nSuccessfully Started\n'
        return json.dumps({'success':True}), 201
    else:
        cursor = db.BaseStation.find({},{'_id':0})

        return dumps(cursor)
```

c) *Mobile phone creation*

```
class mobile():
    def __init__(self):
        self.number = '07' + ''.join(["%s" % random.randint(0, 9) for num in range(0, 9)])
        self.conlimit = random.randint(0, 5)
        self.loc = station.location
```

Database insert and deletion for the mobile phones

```
@app.route('/mobiledata', methods=['POST'])
def mobile_connect():
    data2 = json.loads(request.data)
    db.MobileData.insert_one({
        "stationid":data2['stationid'],
        "number":data2['number'],
        "message":data2['message'],
        "timeofconnection":data2['timeofconnection']
    })

    print str(data2['number'])+" connected to "+str(data2['stationid'])+""
    return json.dumps({'success':True}), 201

@app.route('/mobiledata/<string:id>/<int:number>', methods=['DELETE'])
def mobile_disconnect(id, number):
    #check for existing id in database first
    db.MobileData.delete_one({"number": '0'+str(number),"stationid":str(id)})
    print "Disconnected from "+'0'+str(number)
    return 'Received',200
```

d) *Mobile phone connection and disconnection from the base station*

```
def run(self):
    if self.mode == 'connect':
        while self.run1:
            self.count = check_count()
            if self.gconnect and self.count:
                if self.errq.empty():
                    newconnect = mobile()
                    newconnect.connect('Hello', self.errq)
                    self.q.put(newconnect)
                    time.sleep(random.randint(0,2))
            elif not self.count:
                time.sleep(2)

    if self.mode == 'disconnect':
        time.sleep(8)
        while self.run1:
            while not self.q.empty():
                next_job = self.q.get()
                wait = next_job.conlimit
                if self.quit:
                    wait = 0
                time.sleep(wait)
                #time.sleep(1)
                next_job.disconnect(self.errq)
```

e) *Connection, HTTP and Request error handling, including offline functionality*

```
count = 5
while count:
    try:
        r = requests.post(url, payload)
        r.raise_for_status()
        subt_count()
        break
    except requests.exceptions.ConnectionError as e:
        print e
        print "connection error, fall to send data"
        if dir == 'basestation':
            count -= 1
        else:
            q.put(failreq('post',data,dir))
            time.sleep(5)
            break
    except requests.exceptions.HTTPError as err:
        print err
        print "http requests error",r.status_code
        print "fail to send data"
    except requests.exceptions.RequestException as error:
        print error
        print "Fatal Error Terminating"
        sys.exit(1)
if count == 0:
    print 'Fatal Error Terminating'
    sys.exit(1)
```

8b - Figure 2

Base station database storing

```
@app.route('/basestation', methods=['GET', 'POST'])
def insert_stations():
    if request.method == 'POST':
        data1 = json.loads(request.data)
        db.BaseStation.insert_one({"stationid":data1['id'],
                "stationname":data1['name'],
                "location":data1['location'],
                "capacity":data1['capacity'],
                'timestarted':data1['time'],
                'street':data1['street'],
                'county':data1['county']
        })
```

Base station database deletion

```
@app.route('/basestation/<string:id>', methods=['DELETE'])
def stop_stations(id):
    db.BaseStation.delete_one({"stationid":id})
    print "Base station: "+str(id)+" terminated"
    return 'Received',200
```

8c - Figure 3

a) *Outputting all base station data*

```
data = getdata("basestation")
records = json.loads(data)
#data.split('},')
for i in range(len(records)):
    print "\nStation ID: \t\t%s,\
    \nStation Name: \t\t%s,\
    \nStation Capacity: \t%d,\
    \nStreet: \t\t%s,\
    \nCity: \t\t\t%s,\
    \nCounty: \t\t%s" % (records[i]['stationid'],records[i]['stationname'],records[i]['capacity'],records[i]['street'], records[i]['location'], records[i]['county'])

    print "Time started: \t\t", time.strftime('%Y-%m-%d %H:%M:%S', time.localtime(records[i]['timestarted']))
```

b) *Displaying all base stations at 90% or more of total capacity*

```
data = getdata("basestation/busy")
records = json.loads(data)

for i in range(len(records)):
    print "\nStation ID: \t\t%s,\
    \nStation Name: \t\t%s,\
    \nStation Capacity: \t%d,\
    \nStreet: \t\t%s,\
    \nCity: \t\t\t%s,\
    \nCounty: \t\t%s" % (records[i]['stationid'],records[i]['stationname'],records[i]['capacity'],records[i]['street'], records[i]['location'], records[i]['county'])
```

c) *Displaying the average number of connections over all created base stations*

```
url = "basestation/stat"
choice_data = getdata(url)
records = json.loads(choice_data)
cap = []
for i in range(len(records)):
    print "\nStation ID: \t\t%s,\
    \nStation Name: \t\t%s,\
    \nStation Capacity: \t%d,\
    \nCurrent number of connection: \t\t%s,\
    \nPercentage of used capacity: \t\t%s,\
    " % (records[i]['stationid'],records[i]['stationname'],records[i]['capacity'],records[i]['connected'], records[i]['percent'])
    cap.append(records[i]['connected'])

a = numpy.array(cap)
cmean = numpy.mean(a)
print 'Average number of connection across all base stations is '+str(cmean)+'\n'
```