



# **ASHESI UNIVERSITY COLLEGE**

**An SMS -to- Burro Database Gateway System to  
enable Burro have access to information from their  
Central Database via SMS**

**Applied Project  
B.Sc. Computer Science**

**By  
Momodou K Sowe  
Spring Semester 2017**

**ASHESI UNIVERSITY COLLEGE**

**An SMS -to- Burro Database Gateway System to  
enable Burro to access information from their  
Central Database via SMS**

**Applied Project**

Applied Project submitted to the Computer Science Department, Ashesi  
University College in partial fulfilment of Bachelor of Science Degree in  
Computer Science.

**Momodou K Sowe  
Spring Semester 2017**

## **DECLARATION**

I hereby declare that preparation and presentation of this applied project were supervised in accordance with the guidelines on supervision of applied project laid down by Ashesi University College.

Candidate's Signature:

.....

Candidate's Name:

.....

Date: .....

I hereby declare that preparation and presentation of this Thesis were supervised in accordance with the guidelines on supervision of Undergraduate Thesis laid down by Ashesi University College.

Supervisor's Signature:

.....

Supervisor's Name:

.....

Date:

.....

## Table of Contents

DECLARATION .....	i
Table of Contents .....	ii
Acknowledgement .....	iv
Chapter 1: Title and Introduction.....	1
1.1 Project Overview: .....	1
1.2 Background: .....	1
1.3 Overview of the Marathon System: .....	2
Chapter 2: Requirements.....	4
2.1 Requirements Gathering: .....	4
2.2 Use Cases: .....	5
2.2.1 Marathon Administrator Scenario:.....	5
2.2.2 Marathon Staffer Scenario: .....	5
2.2.3 Marathon Reseller Scenario: .....	6
2.3 Functional Requirements: .....	7
Adding a User .....	7
Deleting user functionality .....	7
Find Marathon users .....	7
The ten (10) different Fodder queries .....	8
Help functionality to tell the users how to structure their Marathon commands and Fodder queries .....	10
Response and Error handling .....	10
Keep records of the communication through the system .....	10
2.4 Non-Functional requirements .....	11
Chapter 3: Architecture and Design.....	12
3.1 High-level System Architecture.....	12
3.2 Software Design:.....	14
3.2.1 Functions Descriptions:.....	14
3.2.2 Flowchart Explanation: .....	19
The following explains the software architecture (fig 2) on page 12: .....	19
Chapter 4: Implementation .....	29
4.1 Hardware.....	29
4.2 Software .....	29
4.3 PostgreSQL Database .....	31
Chapter 5: Tests & Results .....	32

5.1.1 Software Component.....	32
5.1.2 PostgreSQL database component.....	32
5.2 System-level testing .....	33
Fodder Queries:.....	33
Help functionality .....	34
Finding Marathon users Functionality .....	35
Adding User functionality:.....	35
5. Delete User Functionality .....	35
5.4 Analysis of test results .....	36
Chapter 6: Conclusion & Recommendation .....	37
6.1 Project Summary.....	37
6.2 Recommendations.....	37
APPENDICES .....	38
Appendix 1: The re module .....	38
Appendix 2: The String module.....	38
Appendix 4: The pyodbc, time, and serial module .....	39
Appendix 5: The user manual .....	40
Appendix 6: Oliver's Hardware Manual.....	43
Appendix 7: Oliver's Software Manual .....	50
References.....	55

## **Acknowledgement**

I would like to express my special thanks of gratitude to my supervisor, Dr. Buchele, who even though sometimes I acted troublesomely, still effortlessly managed to keep me on my toes and diligently went through my work to point out areas I could do more. She really did turn me into detail oriented individual. I also thank the Burro ICT manager, Mr. Godwill, who gave me the golden opportunity to do this wonderful project on SMS Automated Systems for Burro Brand Ghana Limited. I earnestly thank my project partner, Oliver, for sharing with me the necessary information I needed to effectively pick up where he ended. I am thankful for this new knowledge gained over the course of the project which would not have been otherwise possible without them.

Secondly, my Computer Science lecturers and friends. I sincerely thank them for their aspiring guidance, priceless helpful criticism, and friendly advice related to the project. I am truly grateful to them all for sharing their honest and enlightening views on many issues about this project.

## **Chapter 1: Title and Introduction**

### **1.1 Project Overview:**

Marathon is the nickname of this project. The aim of the project is to automate the process of providing some of the most common information Burro Staffers and hundreds of Burro Resellers throughout Ghana normally call to request from the Burro office. It aims to provide information such as prices, contact info, and discounts from the Burro central database, Fodder, via SMS text message queries even to those in the remotest parts of the country. For example, if a Burro Reseller who resides in a no internet zone wants to look up a certain Burro product's price, the person sends the question in plain language to Marathon's number (MTN sim card in a GSM module sitting in the Burro server room) and Marathon will process this request and sends back the answer right away. Marathon aims to save time and resources for Burro as well as to make this process easier.

### **1.2 Background:**

Burro Brand LLC is a company in Ghana. They sell productivity enhancing products such as solar lights, coal pots, phone chargers, and agricultural equipment. Burro has one office in Koforidua and hundreds of resellers throughout the country. Information like prices, contact information, and discounts is saved in Burro's central database, Fodder. Currently, Burro Staffers and Resellers throughout the country call the office to get information. A lot of time is spent answering phones and looking up information for the caller. As a Burro software engineer intern, I found this process slow and cumbersome and wanted to automate it. An engineering student and fellow Burro intern from Brown University named Oliver Carlsen Moller and I then decided to build Marathon. We divided the project into two parts; Oliver developed version 1 which is the hardware and communication protocols set up, and I developed version 2 which expanded the system to deliver more information to the user and increase the user base (Burro Administrators, Burro Staffers, & Burro Resellers).

### 1.3 Overview of the Marathon System:

Marathon consists of a Raspberry Pi 3 single-board computer, connected to an Adafruit Fona GSM texting module. The Raspberry Pi is set with the Raspbian Linux distribution and other supporting software. This makes the system highly flexible and able to read, send and process SMS text messages while interfacing with other systems such as the Burro SQL server, an internal PostgreSQL Database on the Pi, and the Internet. Python was used to write the drivers and to make the hardware interface. The system uses a standard-sized MTN sim card which resides in the GSM modem.

Marathon, like many other SMS based Information Systems, has four (4) functional features pertaining to query processing; *validation check*, which is used to check if a submitted query is syntactically correct or not; *Parsing and Normalization*, which is used to take out any characters that could cause trouble; *Relevance Check*, which is used to check if the query matches any inbuilt query; and finally the *Query Conversion*, which converts the text to SQL in order to query the server (Joshi & Pathak, 2014). However, unlike many other systems, Marathon uses a GSM modem connection and not an SMS Center (SMSC) of the service providers. This makes Marathon easier to setup and even more cost efficient. Marathon saves Burro cost and the hurdle of finding SMSC of service providers to connect the system. Also, with the MTN SMS bundle service, for instance, the Marathon SIM can currently be bundled for a period of one month as low as GHC1.00 for 50 SMSes whereas, for an SMSC of service providers, a monthly fee or a minimum monthly SMS volume is often required.

Marathon has an internal PostgreSQL database table of trusted numbers. Only phone numbers that are added to this table can use Marathon. Marathon has three (3) different type of users; Burro Administrators, mainly the ICT managers, Burro Staffers, mainly the sales team, and the Burro Resellers. However, to avoid confusions, in relation to Marathon's type of users, I will strictly refer to these as Marathon Administrators, Marathon Staffers, and Marathon



Resellers respectively. The Marathon Administrator has the highest set of privileges, followed by the Marathon Staffer and then the Marathon Reseller. A Marathon Administrator can add or delete any user from the system. A Marathon Administrator and the Marathon Staffer can check all the resellers using the system. The Marathon Resellers have limited access to Fodder's information for security reasons.

Marathon can process 10 different type of Fodder information requests. For example, if a Marathon Staffer wants to find sales details of a reseller, the user sends in "*sales details [ID of reseller]*" to Marathon's number, and it will get you your request, typically in a fraction of seconds. The Marathon query structures are designed to be short to allow ease of use and to minimise errors. Marathon has a "help" functionality to let the user know what the system can do and how to structure his requests. When the user sends "help" to Marathon, it will return: "*help with the marathon command structures or fodder queries?*". Depending on the category the user replies with, the system pulls out all the names and structures of those queries (commands) in that category. The Marathon commands are all the commands use to manage the system such as adding a new user, deleting a user, or finding a Marathon user. They do not interfere with Fodder or return any Fodder information to the user.

This is the goal of Marathon; to automate some of these most common information requests for the Burro community to help them *Do More*, which is what Burro is about.

## Chapter 2: Requirements

### 2.1 Requirements Gathering:

I travelled to Burro to gather the Functional Requirements of the System. Mr. Godwill and I first went through the Marathon version 1.0 Oliver handed over to us to understand what the system can do and to point out areas that needed further development. We recorded down our thoughts and had a video conference call with Oliver. We identified four (4) different areas I will work on; implement the *Fodder Queries*, which were 20 different Burro database information queries; Modify the “*add functionality*” Oliver developed to restrict it to only system’s Administrators; implement *Help functionality*, which aims to look up the different existing Fodder queries and Marathon Commands the system can process; implement *Refill and Check credit balance functionality*, which aims to provide a way the Marathon SIM can be recharged and be able to know Marathon’s credit balance.

However, these functionalities changed over time and new functionalities evolved. During the implementation process, I found out the need to break the system into three (3) different sets of users (Marathon Administrator, Marathon Staffer, & Marathon Reseller) to increase the user base and to enable the Burro Resellers query the some of the Fodder information on their own. This raised some security issues I considered, for instance, a security measure to make sure that no reseller access another reseller’s information. The number of Fodder queries did also reduce to 10, and their query structures changed. Initially, the queries were a bit long, I decided to shorten them to enable the users effectively able to use the system.

Initially, the *add functionality* enables every Marathon user able to add a new user. I modified this functionality to allow only Marathon Administrators able to issue it. I added another Marathon command, delete user functionality, which also only Marathon Administrators can issue. I as well added another Marathon command, find Marathon user,

which finds and return the Marathon users. Detail of these functionalities is provided in section 2.3.

The Help functionality too did change. Instead of retrieving all the different existing queries (commands) at once when as a user sends “help” to Marathon, the system returns *“help with the marathon command structures or fodder queries?”*. Depending on the category the user replies with, the system returns only the queries’ or commands’ names and structures in that category. Section 2.3 provides a detail explanation of its implementation.

## **2.2 Use Cases:**

Marathon has three (3) type of users; a Marathon Administrator, Marathon Staffer, and Marathon Reseller. The following give scenarios of the three (3) different type of users:

### **2.2.1 Marathon Administrator Scenario:**

A new Burro web developer just joined Burro and one of his roles is to oversee the Burro ICT services. The Burro ICT manager, Mr. Godwill, plans to leave the company in a week and wants to hand over his Marathon Administrative powers to this new person to manage the system for Burro. Mr. Godwill will issue the add administrator command to add the web developer to the Marathon as an Administrator, and in turn, the web developer (the new Marathon Administrator) will issue the delete administrator command to delete Mr. Godwill from the system. Practically, Mr. Godwill will send “add admin [name of web developer] with number 024536632” to Marathon and the web developer will be added to the system as an admin. To delete Godwill from the system, the web developer will send “delete admin [phone number of Mr. Godwill]” to Marathon and Mr. Godwill will be deleted from the system.

### **2.2.2 Marathon Staffer Scenario:**

A Burro sales staffer who is currently in the Northern region doing point of purchase (POP) sales on a market day, received an emergency call from his boss at the office that he should

consider making rounds around the region, to collect Burro faulty products from some of their resellers on his way back to the office, for them to fix quickly as the Burro technician just prompted him that he has only 3 work days to go on leave for 2 weeks. Before the Marathon system, his boss has to look up Fodder to communicate the details of this information of all the resellers in the region. With Marathon, the sales staffer can query Fodder via SMS text messaging and immediately get an answer. He will send “reseller [phone number of reseller]” to Marathon to get the ID and name of the reseller. He will then send “job card [ID]” to Marathon to know the details of faulty products associated with the reseller; product name, the status of the product (fixed or not), and the location of the product (with the reseller or at the Burro workshop). This information he will filter and should be able to approach the right resellers on his own for the cost of an SMS without having to call the office.

### **2.2.3 Marathon Reseller Scenario:**

A Burro reseller, Abraham, who resides in the upper east region in a village called Bongo, heard about Burro’s new products and wants to know the prices of some of these products to consider making an order to resell in his village. However, because of the high demand for these products, it is a busy time for Burro; few of their staffers are left in the office to do administrative work and to help with incoming office calls. Many are in the field doing some point of purchase (POP) sales, and some are busy transporting these products to their resellers. Abraham has called the office several times but hasn’t gotten through, and sometimes there is poor network connectivity in the area and he is unable to reach them. With Marathon, Abraham can send “product [name of product]” and will quickly get all the details he needs to know about the product and can decide to order the product or not, for a cost of an SMS.

## 2.3 Functional Requirements:

### Adding a User

Only Marathon Administrators can issue this command. They alone can add new users to Marathon. The add user functionality adds three different type of users; A Marathon Administrator, Marathon Staffer, and Marathon Reseller. After successful user addition, Marathon returns an acknowledgement message to both the adder and the new user. Command structures to add any user are:

add admin *[name]* with number *[number]*

add staffer *[name]* with number *[number]*

add reseller *[name]* with number *[number]*

### Deleting user functionality

Only Marathon Administrators can delete any Marathon user. If Burro decides that a certain Marathon user should no more use the system, this is the functionality they will issue to delete the user from the system. The functionality is designed to be simple, all the Marathon Administrator need to have is the user's number. After successful user deletion, Marathon returns an acknowledgement message to both the Marathon Administrator and the old user. Once a user is deleted, the user can send a text message to Marathon but will not receive a message from Marathon anymore. The commands to delete any type of user are:

delete admin *[number]*

delete staffer *[number]*

delete reseller *[number]*

### Find Marathon users

Only Marathon Administrators can find Marathon Staffers and Marathon Resellers. Marathon Staffers cannot find other Marathon Staffers but only Resellers. Marathon Resellers

cannot find any user. You can find Marathon users in two ways; by searching for all at once, or by their name. The commands to find Marathon users are:

*all staffers*

*staffer named [name of person]*

*all resellers*

*reseller named [name of person]*

### **The ten (10) different Fodder queries**

The following are the different Fodder queries Marathon can process. Marathon Resellers are limited to the first four (4) queries:

#### **1.What is Reseller X's ID?**

Every Marathon user can execute this query, the search is based on reseller's phone number. This query returns a result indicating the reseller's name and ID number, which is used for subsequent queries about the reseller. Marathon Resellers, once they find their IDs, they can ask the following three queries about themselves.

Query structure: *reseller [phone number of reseller]*

#### **2. What is Reseller X's transaction history?**

This query returns answers to the following questions:

- How much does reseller X owe Burro, or what is his balance account?
- When did he last pay Burro, and how much did he pay to Burro, what is the payment receipt no?

*statement [ID of reseller]*

#### **3. What is Reseller X's sales status?**

This query provides answers to the following questions:

- Is Reseller X good, better, or best status?
- What is reseller X paid T90?

*sale status [ID of reseller]*

#### **4. What is the price for product X?**

This query provides answers to following questions:

- What is the Suggested Retail Price (SRP) a product?
- What are the good, better, and best prices for product X?

price *[product name]*

***Only Marathon Administrators and Marathon Staffers can access all the following information:***

#### **5. What is reseller X's details?**

This query provides answers to the following questions:

- What is Reseller X contact details?
- Where is reseller X located?

details *[product name]*

#### **6. What is reseller X's job card?**

This query provides answers to the following questions

- Does Reseller X have any faulty products?
- Have they been fixed?
- Where are they located?

Job card *[ID of reseller]*

#### **7. How many of product X are in the stock?**

Getting total from all stock locations: count *[product name]*

Getting total from specific stock locations: count *[product name, stock location name]*

#### **8. What is the latest note on Reseller X's account?**

Last Note *[ID of reseller]*

#### **9. Who is the last person who contacted Reseller X?**

Who called *[ID of reseller]*

#### **10. What was the last product that was returned by Reseller X?**

Last returned *[ID of reseller]*

## **Help functionality to tell the users how to structure their Marathon commands and Fodder queries**

All Marathon users can issue this command. It returns the list of Fodder queries and Marathon commands the system can process, and their structures. However, to reduce cost, these queries and commands are grouped into two categories, “help with fodder queries”, and “help with Marathon commands”. If a user sends in “help”, the system will return these two categories from which the user will decide which information s/he needs. If for instance, the user is interested in the Marathon commands, s/he will send in “*help with marathon command*” and the system will return the Marathon commands. The same applies when s/he sends “help with fodder queries”. The following are commands to get help with Marathon:

*“help”*

*“help with fodder queries”*

*“help with Marathon commands”*

## **Response and Error handling**

The system will return appropriate acknowledgement messages to the user(s). For example, when a user is added, Marathon will send a welcome message to the newly added Marathon user, and another text message to the Marathon Administrator, acknowledging that the new user is successfully added. In the case of errors, Marathon will send back an error message to the Marathon Administrator.

## **Keep records of the communication through the system**

Marathon keeps a log of all the transaction that happens in the system. For instance, if a user is added to Marathon, the system keeps a log of who added who, and if there were errors, it logged them before it sends a response to the user. Oliver developed this functional requirement.



## **2.4 Non-Functional requirements**

- (i) The user must have a telephone that sends and receives SMS to be able to use Marathon
- (ii) The user must have a Ghana phone number to be able to use the system.
- (iii) The user and the Marathon SIM must have credit units in order to be able to send and receive SMS from Marathon respectively.
- (iv) The user must be in a network coverage area to be able to send and receive SMS text messages from Marathon
- (v) The intranet at the Burro office must be up for the system to work

## Chapter 3: Architecture and Design

### 3.1 High-level System Architecture

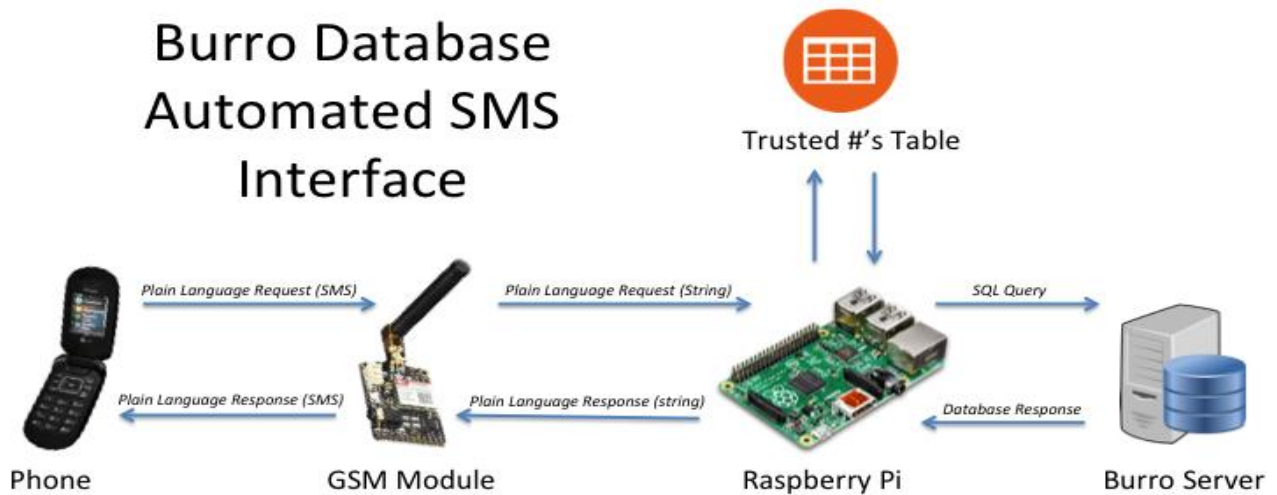


Fig 1

The following is a high-level architecture of how the system works

- A user sends a text with some query like "reseller 0243405663"
- The GSM module receives this text
- The GSM module passes the text to the Raspberry Pi, along with the phone number of the sender
- The Raspberry Pi checks the phone number of the sender against an internal database of trusted phone numbers to make sure the sender can access the system. If the phone number is good, then it passes the message along to a Python script.
- The Python script in the Raspberry Pi picks up the text message and parses it, looking for key phrases to see what the user wants. In this case, it recognises the phrase "reseller", meaning that the user wants to look up the ID of a Burro Reseller.
- The python script logs onto the Burro Server, using an SQL connection and looks up the reseller with number 0243405663
- Having received the answer, the script puts together a text and passes it to the phone module. In this case: "RESULT: ID – 1127 Name - Momodou K Sowe",
- The phone module receives the text and puts it in a text message to be sent back to the sender.
- The sender receives an answer in typically just seconds.

## Marathon 2.0 Software Architecture

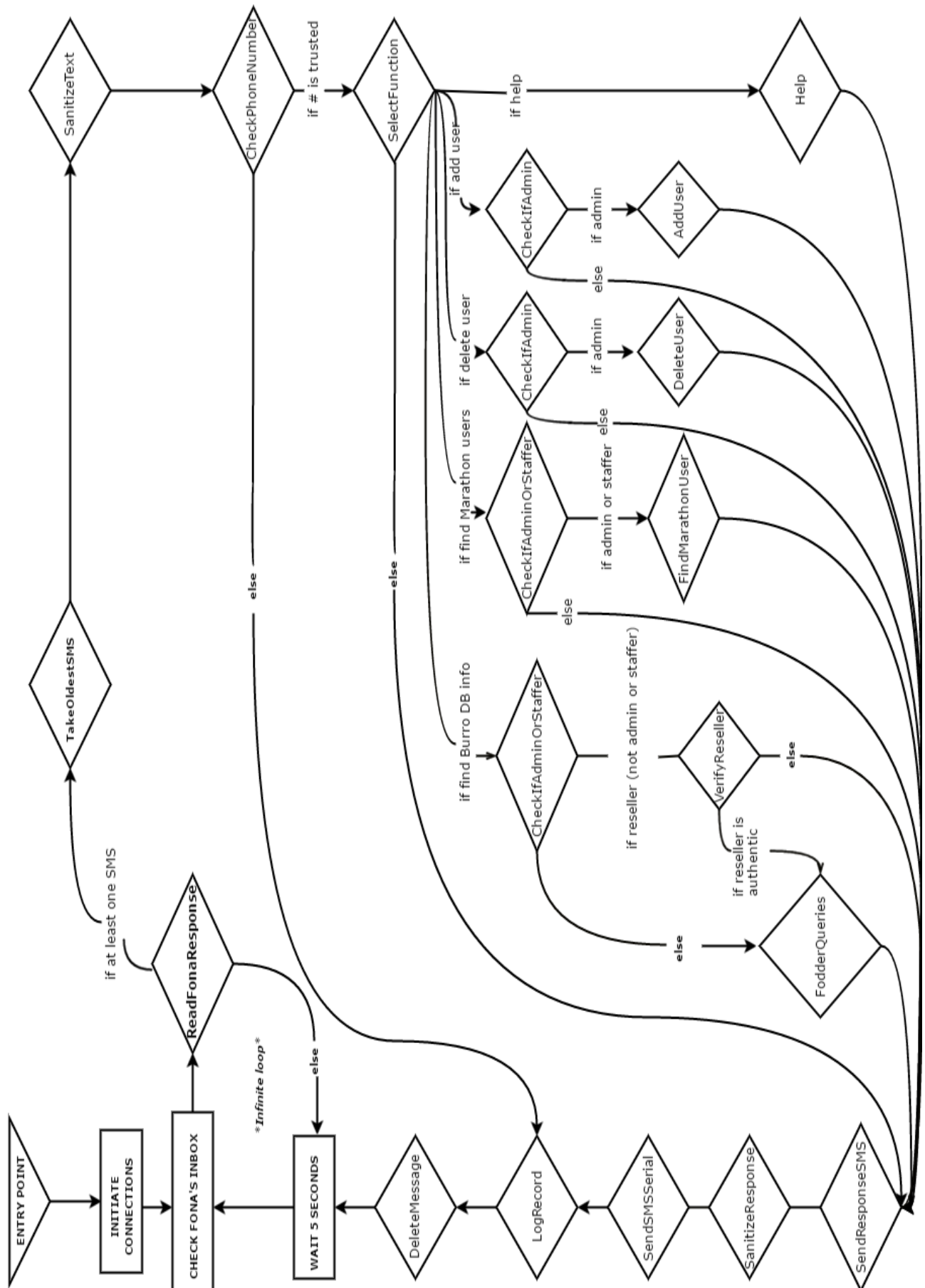


Fig 2

### 3.2 Software Design:

Marathon has many functions defined to simplify the code and to enable code reuse. The Marathon program is broken down into four (4) scripts. One of the scripts (Marathon.py) is the main script that runs infinitely in the background when the program starts. It initiates the hardware connection protocols. It connects Marathon to the Burro Server, the GSM module, and the internal PostgreSQL database. There is another script called the MarathonFunction.py. This script mainly contains a class called MarathonFunctions which defines all the functions that treat the SMS. Another script is the MarathonCommand.py. This script contains all the Marathon commands. Another script is the Fodder.py which contains all the Fodder queries.

The MarathonFunctions class in the MarathonFunctions.py script contains the following functions. The first seven (7) and the ninth (9<sup>th</sup>) function were developed by Oliver.

#### 3.2.1 Functions Descriptions:

**1. ReadFonaResponse:** This function reads the response string from the Fona (GSM Module). If there are no text messages, the function goes into waiting for a five (5) seconds before calling on the system to check the inbox again. If there are text messages, the function passes them on to a function called TakeOldestSMS.

**2. TakeOldestSMS:** This function selects the oldest SMS in the list passed by ReadFonaResponse. The oldest SMS is the one that reaches the Fona's inbox first. This oldest SMS is then passed to another function called SanitizedText.

**3. SanitizedText:** This function takes out any characters such as apostrophes, double quotes, semicolons from the SMS that could cause trouble for instance when logging the text in the PostgreSQL database, and when looking for key phrases to see what the user wants. After the text is sanitised, it is then passed to another function called CheckPhoneNumber.

**4. CheckPhoneNumber:** Marathon has internal PostgreSQL database which stores all the Marathon users' numbers and names. This function checks the phone number in the current SMS against the table of trusted numbers from this database.

**5. SelectFunction:** This function figures out which functionality the user is requesting. It matches the text message with inbuilt string to decide which functionality to execute. Depending on the type of functionality the user wants to execute, it either calls that particular functionality to execute directly or calls another function to first check if the user is allowed access. I added the latter to tighten security measures. Regardless of the functionality the user wants to execute, the SendResponseSMS always got called at the end to send back the response to the user.

**6. SendResponseSMS:** This function sends a response back to the user. This function calls two other functions, SendSMSSerial, and SanitizeResponse.

**7. SendSMSSerial:** This function handles the serial port communication (GSM module) necessary to send an SMS. It delivers the response to the user through the GSM module. The SendSMSSerial also calls the SanitizeResponse function.

**8. SanitizeResponse:** This function removes the characters such as apostrophes, semicolons, quotation marks that could cause trouble when logging the response into the Records table. This function also calls the LogRecord function to log the record of the transaction. I added this function.

**9. LogRecord:** The Marathon's internal PostgreSQL database has a table called Records. This function logs a record of the transaction in that table. It stores the sender's number, sender's message, sender's response, and whether the sender's number is trusted.

**I added the following functions:**

**10. CheckIfAdmin:** There are Marathon commands limited to only Marathon Administrators such as adding a user and deleting user. This function checks to find if the sender (phone number) is a Marathon Administrator before the system can execute this command.

**11. CheckIfAdminOrStaffer:** There is a Marathon command only Marathon Administrators and Marathon Staffers can issue which is finding the Marathon resellers. This function checks to confirm that the sender is a Marathon Administrator or a Marathon Staffer before the command is executed.

**12. VerifyReseller:** Since there are information resellers can also find about themselves, this function ensures that each reseller is accessing his only information.

The MarathonCommands.py contains the following functions:

**13. AddUser:** There is no function in the code called AddUser but AddAdmin, AddStaffer, and AddReseller. I used AddUser purposely to simply the diagram. There are tables in the internal PostgreSQL database I created called AdminNumbers, StaffNumbers, ResellersNumbers. The AddAdmin function adds a user to Marathon as an Administrator by storing the user's number in the AdminNumbers, the AddStaffer adds the user as a Marathon Staffer and stores the number in StaffNumbers. The AddReseller adds the user as a Marathon Reseller and stores the number in the ResellersNumbers. All the numbers of these different type of Marathon users are as well kept in a general table called trustedNumbers in the PostgreSQL database Oliver created.

**14. DeleteUser:** There is no function in the code called DeleteUser but DeleteAdmin, DeleteStaffer, and DeleteReseller. The DeleteAdmin function deletes a Marathon

Administrator, the DeleteStaffer deletes a Marathon Staffer, and the DeleteReseller deletes a Marathon Reseller from Marathon. Deleting a user means deleting his number from the trustedNumbers as well as from the user's specific table.

**15. FindMarathonUser:** This function finds all the types of Marathon users except Marathon Administrators. Only Marathon Administrators and Marathon Staffers can find a Marathon user. Marathon Administrators can find Marathon Staffers and Marathon Resellers. Marathon Staffers on the other hands can only find Marathon Resellers.

**16. Help:** There is a table called Help in the internal PostgreSQL database. All the Marathon commands and Fodder queries' structures are stored in this table. This function finds all the (8) different Marathon commands and the (10) different Fodder queries Marathon can process: The Marathon commands are *the AddAdmin, AddStaffer, AddReseller, DeleteAdmin, DeleteStaffer, DeteleReseller, FindStaffer, and FindReseller*. When a user sends in *help*, it returns "help with the command structures or fodder queries?". If the user sends in *marathon commands*, Marathon will return all the Marathon command structures. On the other hands, if the user sends in *fodder queries*, Marathon will return all the fodder queries. The list of Fodder queries are given below.

All the following function are in the Fodder.py script

**FodderQueries:** This function is not in the code. I used the name intentionally to cover up the ten (10) different fodder queries Marathon can process. I have, however, explicitly added these queries below. Most of the Fodder queries use the ID of the reseller as the search criteria and as such, the FindID function provides the entry for most of the queries. The Burro Reseller's information access is limited to the first four (4) functions.

**17. FindID:** This function looks up the ID of a specific reseller. When an authenticated user sends in *reseller [phone number of reseller]*, this function checks the ID of the reseller in the Burro's central database, Fodder. The phone number is the search criteria and it returns the ID and name of the reseller.

**18. Statement:** This function returns the transaction history associated with a reseller. It provides answers to question like how much a reseller owe, what is the reseller account balance, when did the reseller last pay, how much did the reseller pay, what is the reseller's receipt number. This function got called when an authenticated user sends in "*statement [ID of reseller]*". It uses the reseller's ID as the search criteria.

**19. SalesStaus:** It looks up the sales status of a specific reseller. When an authenticated user sends in *sales status [ID of reseller]*, this function checks in Fodder and returns T90, discount level, and credit limit associated to the reseller. It uses the reseller's ID as the search criteria.

**20. Price:** This function returns suggested retail price (SRP) of a product and informs the sender the good, better, and best prices for a product. It got called when an authenticated user sends in *price [product name]*. It uses the product name as the search criteria.

The following functions are limited to only Marathon Administrators and Burro Staffers:

**21. FindDetails:** This function looks up the details of a specific reseller. When either of these two users (Marathon Administrators and Marathon Staffers) sends in *details [ID of reseller]*, this functions will return the reseller's residence area, landmark, and region from Fodder. The reseller's ID is the search criteria.

**22. JobCard:** When either of the users sends in *job card [ID of reseller]*, this is the function that got called and returns from Fodder the product name, the status of the product (whether



faulty or not), and location of the product associated to the reseller. The reseller's ID is the search criteria.

**23. LastNote:** This function provides answers to this question: what is the latest note on the reseller's account. When either of the users sends in *last note [ID of reseller]*, this function is called and returns from Fodder only the latest notes on the reseller's account. The ID is the search criteria.

**24. WhoCalled:** This function provides answers to this question: who is the last person who contacted reseller X? The function only returns the caller's name from Fodder. The reseller's ID is the search criteria.

**25. Last returned:** Looks up the last product returned by a reseller. This function is called when either of the users sends in *last returned [ID of reseller]*. The function only returns the product name from Fodder. The reseller's ID is the search criteria.

**26. Count:** This function informs either of the users how many of product X are in the stock and how many of product X are in a specific stock location. If either of the users sends in *count [product name]*, the function returns the total amount of that product in the stock. If the user sends in *count [product name, stock name]*, the function returns the total amount of that product in that stock. In the former, the product name is search criteria, and in the latter, it is both the product name and the stock name.

### 3.2.2 Flowchart Explanation:

The following explains the software architecture (fig 2) on page 12:

- When the program starts, the Marathon.py script initiates the hardware connections. It connects to Fodder, to the internal PostgreSQL database, and to the GSM module.

- The main Python file is a small script that runs in an infinite loop checking if there is an SMS in the GSM module (Fona's inbox). This script calls the **ReadFonaResponse** function from the MarathonFunctions.py script which reads the response from the Fona. If there is no text message, it waits for **5 seconds** and reads again and if there is a text message the **ReadFonaResponse** function passes the text message to another class function of the MarathonFunctions.py script called **TakeOldestSMS**
- The **TakeOldestSMS** function takes the oldest SMS which is the first SMS in the list (that is the first to reach to the Fona) and forwards it to the **SanitizeText** class function which takes out any characters from the SMS that could cause trouble. It removes all the characters such as commas, apostrophes, double quotes, semicolons from the text message so that it is only a plain text (string).
- Before the text message can carry out any function, the sender's number is checked against an internal PostgreSQL table of trusted Numbers. This is done by another class function called **CheckPhoneNumber**. If the number is not trusted, the function calls another class function, **LogRecord**, to logs the record of the transaction (that is the sender's number, text message, response, and whether the number is trusted) in the records table and the sender does not get any response back. The sender's message is as well deleted from the GSM module (that is the sender's number and text message). Else if the number is trusted, the message is forwarded to another class function called **SelectFunction**.
- The **SelectFunction** is a big function and determines what the sender wants to carry out. It checks the message (string) the user sent in against internal built-in queries to determine which to execute. If the sender sends a message that matches any of the inbuilt queries this function will forward the message to that function that handles that particular type of query.

- If the sender sends in “*help*”, The SelectFunction forwards the string to the **Help** function in MarathonCommands.py. This function returns “*help with the marathon command structures or fodder queries?*”. Depending on the user’s reply, the function pulls out either all the Marathon commands or the Fodder query structures and send them back to the user through the **SendResponseSMS** function.
- If the sender sends in “*add admin Dr. Buchele with number 0556427918*”, a class function, **CheckIfAdmin**, is called to check if the sender is Marathon Administrator in the AdminNumbers table. If s/he is not, the CheckIfAdmin function calls another class function, **SendResponseSMS**, to return “*this number is not an admin*” as a response to the user. Else, the CheckIfAdmin function forwards the string to the AddAdmin function in the MarathonCommand.py script. The AddAdmin function stores Dr. Buchele’s name and number both in the AdminNumbers table and in the trustedNumbers table. The function then sends a welcome message (*Welcome New Marathon Admin! Send HELP to get help*) to the new Marathon Administrator through an inherited function, the SendResponseSMS function from the MarathonFunction.py script.
- If the sender sends in “*add staffer Mr. Godwill with number 0278761893*”, the **CheckIfAdmin** function is called to check if the sender is a Marathon Administrator. If s/he is not, the CheckIfAdmin function calls the **SendResponseSMS** function to send back the response as above to the sender. Else, the CheckIfAdmin function forwards the string to the AddStaffer function in the MarathonCommand.py script. The AddStaffer function stores Mr. Godwill’s name and number both in the StaffNumbers table and in the trustedNumbers table. The function then sends a welcome message (*Welcome New Marathon Staffer! Send*

*HELP to get help*) to the new Marathon Staffer through the SendResponseSMS function.

- If the sender sends in “*add reseller Momodou with number 0256436663*”, the **CheckIfAdmin** function is called to check if the sender is a Marathon Administrator. If s/he is not, the CheckIfAdmin function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdmin function forwards the string to the AddStaffer function in the MarathonCommand.py script. The AddStaffer function stores Momodou’s name and number both in the ResellersNumbers table and in the trustedNumbers table. The function then sends a welcome message (*Welcome New Marathon Reseller! Send HELP to get help*) to the new Marathon Reseller through the SendResponseSMS function.
- If the sender sends in “*delete admin 0556427918*”, the **CheckIfAdmin** function is called to check if the sender is a Marathon Administrator. If s/he is not, the CheckIfAdmin function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdmin function forwards the string to the DeleteAdmin function in the MarathonCommand.py script. The DeleteAdmin function deletes Dr.Buchele’s name and number both in the AdminNumbers table and in the trustedNumbers table. The function then sends a message (*You are deleted from Marathon!*) to Dr. Buchele and sender through the SendResponseSMS function.
- If the sender sends in “*delete staffer 0278761893*”, the **CheckIfAdmin** function is called to check if the sender is a Marathon Administrator. If s/he is not, the CheckIfAdmin function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdmin function forwards the string to the DeleteStaffer function in the MarathonCommand.py script. The DeleteStaffer function deletes Mr. Godwill’s name and number both in the StaffNumbers table and in the

trustedNumbers table. The function then sends a message to Mr. Godwill and the sender through the SendResponseSMS function.

- If the sender sends in “*delete reseller 0256436663*”, the **CheckIfAdmin** function is called to check if the sender is a Marathon Administrator. If s/he is not, the CheckIfAdmin function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdmin function forwards the string to the DeleteReseller function in the MarathonCommand.py script. The DeleteReseller function deletes Momodou’s name and number both in the ResellersNumbers table and in the trustedNumbers table. The function then sends message (*You are deleted from Marathon!*) to Momdou and the sender through the SendResponseSMS function
- If the sender sends in “*all staffers*”, the **CheckIfAdmin** function is called to check if the sender is a Marathon Administrator. If s/he is not, the CheckIfAdmin function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdmin function forwards the string to the **MarathonUsers** function in the MarathonCommand.py script. This function returns all the Marathon Staffers (name and phone numbers) in the StaffNumbers table. This result is sent back to the sender through the SendResponseSMS function. In case there is no result, the sender receives “*Sorry, your search gave no results*”.
- If the sender sends in “*staffer named Mr. Godwill*”, the **CheckIfAdmin** function is called to check if the sender is a Marathon Administrator. If s/he is not, the CheckIfAdmin function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdmin function forwards the string to the **MarathonUsers** function in the MarathonCommand.py script. This function searches “*Mr. Godwill*” in the StaffNumbers table. The response is sent back to the sender through the SendResponseSMS function.

- If the sender sends in “*all resellers*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Staffer in the AdminNumbers and StaffNumbers table. If s/he is not either of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response (“*The number is neither a staffer nor an Admin!*”) to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **MarathonUsers** function in the MarathonCommand.py script. This function returns all the Marathon Resellers (name and phone numbers) in the Resellers.Numbers table. The response is sent back to the sender through the SendResponseSMS function.
- If the sender sends in “*reseller named Momodou*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If s/he is not any of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **MarathonUsers** function in the MarathonCommand.py script. This function searches “*Momodou*” in the ResellersNumbers table. The response is sent back to the sender through the SendResponseSMS function.
- If the sender sends in “*details [ID of reseller]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If s/he is not any of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **Details** function in the Fodder.py script. This function will return the reseller’s *residence area, landmark, and region*. In case there is no result, “*Sorry, your search gave no result*” is sent back to the sender through the SendResponseSMS function.

- If the sender sends in “*who called [ID of reseller]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If s/he is not any of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **WhoCalled** function in the Fodder.py script. This function will return the caller. In case there is no result, “*Sorry, your search gave no result*” is sent back to the sender through the SendResponseSMS function.
- If the sender sends in “*last returned [ID of reseller]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If s/he is not any of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **LastReturned** function in the Fodder.py script. This function will return the products returned. In case there is no result, “*Sorry, your search gave no result*” is sent back to the sender through the SendResponseSMS function.
- If the sender sends in “*last note [ID of reseller]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If s/he is not any of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **LastNote** function in the Fodder.py script. This function will return the last notes on the reseller’s account. In case there is no result, “*Sorry, your search gave no result*” is sent back to the sender through the SendResponseSMS function.

- If the sender sends in “*count [product name]*” or count “*[product name in stock name]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If s/he is not any of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **Count** function in the Fodder.py script. This function will either return the number of product X in the stock or the number of product X are in a specific stock location depending on the query the sender sends to Marathon. In case there is no result, “*Sorry, your search gave no result*” is sent back to the sender through the SendResponseSMS function.
- If the sender sends in “*job card [ID of reseller]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If s/he is not any of these, the CheckIfAdminOrStaffer function calls the SendResponseSMS function to send back the response to the sender. Else, the CheckIfAdminOrStaffer function forwards the string to the **JobCard** function in the Fodder.py script. This function returns the product name, the status of the product (whether faulty or not), and location of the product associated with the reseller. In case there is no result, “*Sorry, your search gave no result*” is sent back to the sender through the SendResponseSMS function.
- If the sender sends in “*reseller 0234400433*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If the sender is a Marathon Reseller or Marathon Administrator, the CheckIfAdminOrStaffer function forwards the string directly to the **FindID** function of the Fodder.py script. This function searches “0234400433” in Fodder and returns the reseller’s name and ID associated with it. The result is sent back to the sender through the SendResponseSMS



function and if there is no result, *“Sorry, your search gave no result”*, is sent back to the sender. If the sender is neither a Marathon Administrator nor a Marathon Staffer, that means that the sender is a reseller and the CheckIfAdminOrStaffer function calls the **VerifyReseller** function to check in the ResellersNumbers table to confirm that the sender’s (Marathon Reseller) phone number is the same as the *“0234400433”*. If the numbers are not the same, it calls the SendResponseSMS function to send back *“The number you provided is not the same as your number”* to the sender, and if the numbers are the same, the VerifyReseller function forwards the string to the **FindID** function in the Fodder.py script.

- If the sender sends in *“statement [ID of reseller]”*, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If the sender is a Marathon Reseller or Marathon Administrator, the CheckIfAdminOrStaffer function forwards the string directly to the **Statement** function of the Fodder.py script. This function will return the reseller’s *Account Balance, Last Payment made, Date Paid, and Payment Receipt Number* from Fodder. The result is sent back to the sender through the SendResponseSMS function and if there is no result, *“Sorry, your search gave no result”*, is sent back. If the sender is not a Marathon Staffer or Marathon Administrator, the CheckIfAdminOrStaffer function calls the **VerifyReseller** function to confirm the Reseller. It first searches the ID of the reseller in Fodder using the sender’s phone number (SIM number) as the search criteria. It then checks to confirm that the provided reseller ID is the same as the returned ID. If the IDs are not the same, it calls the SendResponseSMS function to send back *“The ID you provided is not the same as your ID”* to the sender, and if the IDs are the same, the VerifyReseller function forwards the string to the **Statement** function in the Fodder.py script.

- If the sender sends in “*sales status [ID of reseller]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If the sender is a Marathon Reseller or Marathon Administrator, the CheckIfAdminOrStaffer function forwards the string directly to the **Sales Status** function of the Fodder.py script. This function will return the reseller’s *Paid T90, Discount Level, and Credit Limit* from Fodder. The response is sent back to the sender through the SendResponseSMS function. If the sender is a Marathon Reseller, the **VerifyReseller** function is called to search the ID of the reseller in Fodder using the sender’s SIM number as the search criteria. It then checks to confirm that the provided ID reseller is the same as the returned ID. If the IDs are not the same, it calls the SendResponseSMS function to send back “*The ID you provided is not the same as your number*” to the sender, and if the IDs are the same, the VerifyReseller function forwards the string to the **SalesStatus** function in the Fodder.py script.
- If the sender sends in “*price [name of product]*”, the **CheckIfAdminOrStaffer** function is called to check if the sender is a Marathon Administrator or Marathon Staffer. If the sender is a Marathon Reseller or Marathon Administrator, the CheckIfAdminOrStaffer function forwards the string directly to the **Price** function of the Fodder.py script. This function will return the product’s suggested retail price (SRP), the product’s good, better, and best price from Fodder. In case there is no result, “*Sorry, your search gave no result*” is sent back to the sender through the SendResponseSMS function. Though this price function is one of the functions Burro Resellers can use, it does not pull out any information associated with the Burro Reseller, thus, there is no need to verify the sender as a reseller. If the sender is neither a Marathon Administrator nor a Marathon Staffer. The **Selection** function directly forwards the string to the **SalesStatus** function in the Fodder.py script.

## Chapter 4: Implementation

### 4.1 Hardware

The Marathon hardware consists of a Raspberry Pi 3.0 and an Adafruit Fona GSM modem as well as an antenna and a battery used to deal with power interruption. All software is stored on an 8GB micro-SD card in the Pi. The physical system fits inside a 3D printed case and has been sent to Burro's offices in Koforidua where it sits in the server room. To learn more about the hardware, I have attached the *Marathon Hardware Setup Manual Oliver developed at Appendix 7*.



### 4.2 Software

The system was developed to run on the Linux distribution NOOBS 2.1, based on Raspbian. The script was written for Python 2.7 with the Anaconda package. The script is started automatically on reboot using Crontab. More information on how the system was set up is also available in the *software setup manual Oliver developed at Appendix 8*.

The Python script that Oliver had all the functions and variables in one file. I broke the Marathon program (the Python script) into four (4) scripts. One of the scripts (Marathon.py) is the main script that runs infinitely in the background when the program starts. It initiates the hardware connection protocols. It connects Marathon to the Burro Server, the GSM module,

and the internal PostgreSQL database. There is another script called the MarathonFunction.py where I defined a class called MarathonFunctions. This class contains all the functions that treat the SMS. It contains mainly the functions Oliver developed and the few ones I added such as the SanitizeResponse, CheckIfAdmin, CheckIfAdminOrReseller, and the VerifyReseller function. This class, all the other scripts imports it to be able to reuse its class variables or methods. The Marathon.py script imports it to be able to reuse its class variable and methods such as the *marathonConnection*, *fodderConnection*, *serialConnection* variables, and *ReadFonaResponse* function. When the Marathon.py script reads an SMS from the Fona, it passes this SMS to the ReadFonaResponse function which calls series of functions in its class to treat the SMS before it passes it on to either the MarathonCommand.py, or Fodder.py to process the user's request. The MarathonCommand.py contains all the Marathon commands such as the AddAdmin, AddStaffer, AddReseller, FindMarathonUser, and the Help function. The Fodder.py contains all the Fodder queries. It also imports the MarathonFunctions class to be able to send back responses to the user and log the transaction through the SendResponseSMS function.

The Python packages I used are re, String, math, pyodbc, time, and the serial module. The **re** module is what is used to regex the incoming message against the inbuilt strings to know the right function to call. The is shown in appendix 1. The **String** module is used to sanitise the response from Fodder, to take out any characters that could cause trouble when logging the transaction in the Records table. It is shown in Appendix 2. The **math** module is used compute how many blocks a message need to be break into if it exceeds 160 characters as shown in Appendix 3. The **time module** is what is used to program the main script to check the GSM module for new message after every five (5) seconds. The **pyodbc module** is what is used to connect to the internal PostgreSQL database and the Burro Server. The **serial module** is what is used to connect to the GSM module. These three (3) are shown in Appendix 4.

### 4.3 PostgreSQL Database

Marathon uses a small, locally hosted Postgres database to keep track of users, to keep track of the different Marathon commands and Fodder queries, and to store a log of all communication through the system. Oliver created this database which had only two tables, the Record and the trustedNumbers table. I added the AdminNumbers table, the ResellersNumber table, the StaffNumbers table, and the Help table. The structures of these tables are provided below:

```
TrustedNumbers (  
    ID serial primary key,  
    Number varchar (30) NOT NULL,  
    Name varchar (60) NOT NULL  
  
    );  
  
AdminNumbers (  
    ID serial primary key,  
    Number varchar (30) NOT NULL,  
    Name varchar (60) NOT NULL  
  
    );  
  
StaffNumbers (  
    ID serial primary key,  
    Number varchar (30) NOT NULL,  
    Name varchar (60) NOT NULL  
  
    );  
  
ResellersNumbers (  
    ID serial primary key,  
    Number varchar (30) NOT NULL,  
    Name varchar (60) NOT NULL  
  
    );  
  
Help (  
    ID serial primary key,  
    Query varchar (200) NOT NULL,  
    QueryStructure varchar (200) NOT NULL  
  
    );  
  
Records (  
    ID serial primary key,  
    Number varchar (30) NOT NULL,  
    Message varchar (160),  
    Response varchar (200),  
    Trusted varchar (1)  
  
    );
```

## Chapter 5: Tests & Results

I divided the test into three sections; the component test results, the system-level testing results, and the user testing results.

### 5.1 Component testing

#### 5.1.1 Software Component

For every Fodder query I implemented, I executed it locally to make sure that the SQL syntax is correct, and to know the nature of its output in order to present the information in the best possible format to the user. The following screenshot shows a sample Fodder query I tested from my own PC (locally):

```
Connecting to BuroDB..
Connected
RESULT: productName: Solar Eco Light - jobStatus: Fixed - productLocation: Workshop ;
productName: Battery Charger - jobStatus: Beyond Repairs - productLocation: Workshop ;
```

When I send in a message and fail to receive a response from the system, I log on to the Pi to see what goes wrong at the backend. This helps me to debug my code more quickly. The following screenshot shows a sample output of a Burro reseller's ID I requested, running on the Pi:

```
There are new text messages
Fetching the oldest unread text message
Oldest unread text message fetched:
['2', '+233243405663', 'reseller 0266402741']
Sanitizing message text
Message text sanitized
The number is trusted
Requesting to find a reseller
The number is not a reseller but a trusted number
----- SMS -----
RESULT: ID 11218 - Christiana Kumah Siaw ; ID 15056 - God's Time is the Best ;
----- SMS -----
```

#### 5.1.2 PostgreSQL database component

One of the components of the system is the *PostgreSQL database*. The different type of users each has a separate table. To create a table on the Pi, I log in to the Pi and issue some Linux commands to create it and test it. Initially, I add a user directly to make sure that the query is

right, and then I will try to output the result. The following screenshots show a sample output of some of the different type of users (tables) I tested componentry.

Marathon Administrator:

```
Marathon=# select * from AdminNumbers;
id |      number      | name
----+-----+-----
  6 | +233246122913 | hussain
  7 | +233243405663 | momodou
(2 rows)
```

Marathon Staffer:

```
Marathon=# select * from StaffNumbers;
id |      number      | name
----+-----+-----
  1 | +233248664660 | Koby
(1 row)
```

Marathon Reseller:

```
Marathon=# select * from ResellersNumbers;
id |      number      | name
----+-----+-----
  1 | +233248664660 | Koby
(1 row)
```

## 5.2 System-level testing

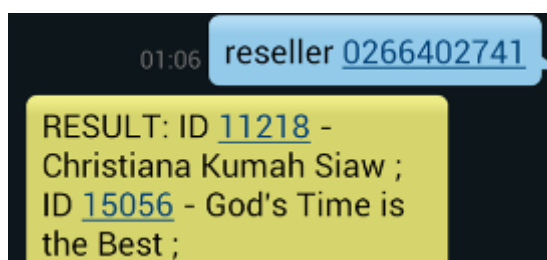
### Fodder Queries:

I implemented all the ten (10) Fodder queries. However, in this section I am going to show four only (3) different Fodder queries test results:

(1) What is my Reseller ID?

The sender sends in “reseller [phone number]” and Marathon will give the reseller’s ID and name.

For example,



The screenshot shows a chat interface with a dark background. At the top, there is a timestamp '01:06' and a blue speech bubble containing the text 'reseller 0266402741'. Below this, a yellow rectangular box contains the response text: 'RESULT: ID 11218 - Christiana Kumah Siaw ; ID 15056 - God's Time is the Best ;'. The IDs '11218' and '15056' are highlighted in blue.

(2) What is my contact details?

The sender sends in “details [reseller ID]” and Marathon will give the reseller’s phone number, his residence, landmark, and region.

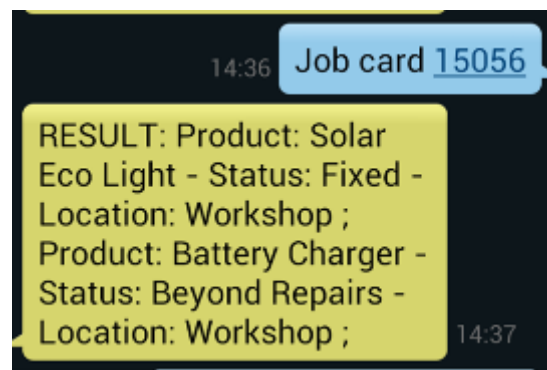
For example,



(3) What is my job card history?

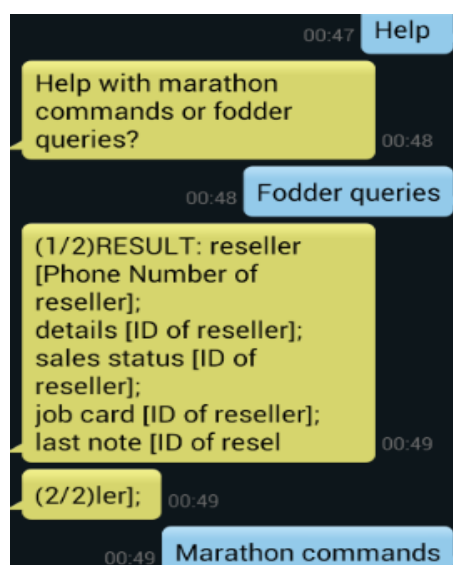
The sender sends in “job card [reseller ID]”

For example,



## Help functionality

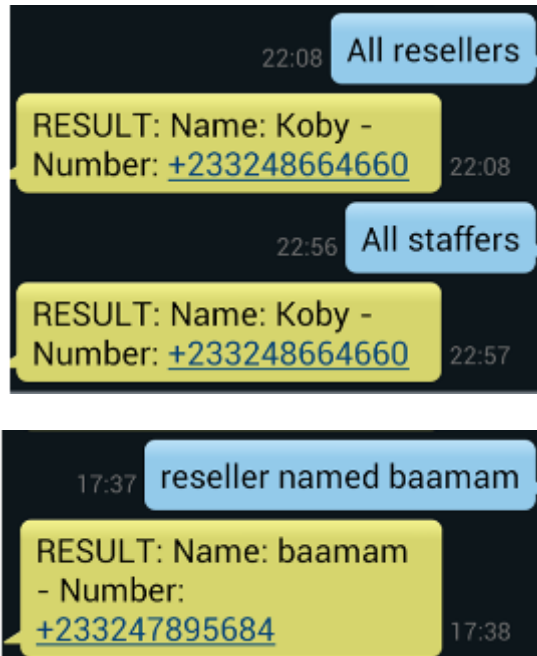
Getting help on how to structure Fodder queries and Marathon Commands structures:





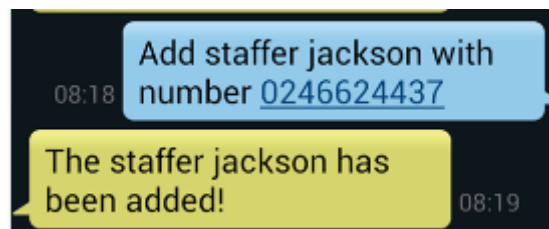
## Finding Marathon users Functionality

Finding out people using Marathon:

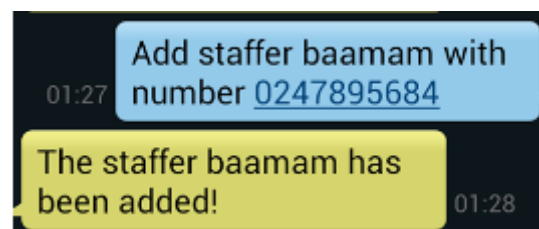


## Adding User functionality:

Adding a Marathon Staffer:

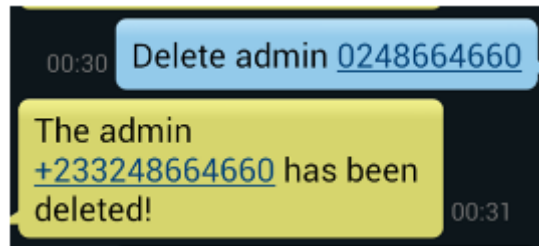


Adding a Marathon Reseller:

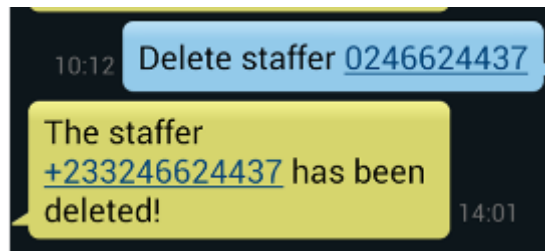


## 5. Delete User Functionality

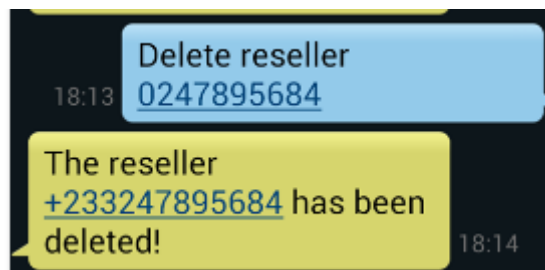
Deleting an Administrator:



Deleting a Marathon Staffer:



Deleting a Marathon Reseller:



#### 5.4 Analysis of test results

Marathon works fine all the time typically. However, there are instances where there are connection errors to the modem due to network connection errors and the AT commands get included in the SMS sent out to the user. Apart from this intermittent error, it pulls out the right records from both databases. The system is not forgiving with errors in your questions. If a user sends a message to Marathon with even a single syntax error in the query, Marathon cannot understand what the user wants and simply returns *"Sorry, I don't understand your request."*

## **Chapter 6: Conclusion & Recommendation**

### **6.1 Project Summary**

Marathon aims to save the Burro community time and it is designed to be very simple to use. The project aimed to enable Burro Staffers and Resellers access the information they need the most outside the office from the Burro Central Database via SMS text messaging. Marathon made it possible for Burro resellers throughout the country to have access to information like their transaction history or statement (that is their Account Balance, Last Payment, Date Paid, and Payment Receipt No) from the Burro Central Database via SMS text message for the cost of an SMS.

### **6.2 Recommendations**

The project has been successful in setting up the basic infrastructure and expanding its functionality to meet the needs of Burro staff and resellers in the field, I, however, recommend the following potential areas for improvement:

- Set up automatic recharging of the credit for the Marathon SIM card
- Implement some automated way of checking if there is credit on the Marathon SIM
- Implement some way of checking if Marathon is running properly without having to SSH into the system
- Implement Natural language processing to increase the flexibility of Marathon. For instance, if a sender makes errors in their questions, the system guess the query intended and response with the right result.

## APPENDICES

### Appendix 1: The re module

The **re** module is what is used to regex the in incoming message against the inbuilt strings to know the right function to call.

```
def SelectFunction(CurrentSMS):
    # Figures out which function the user is requesting

    currentSMSText = CurrentSMS[2]
    if re.match('add staffer ',currentSMSText.lower()) != None:
        PrintStatusMessage('Requesting to add staffer')
        CheckIfAdmin(CurrentSMS)
    if re.match('delete staffer ',currentSMSText.lower()) != None:
        PrintStatusMessage('Requesting to delete staff')
        CheckIfAdmin(CurrentSMS)
    if re.match('add admin ',currentSMSText.lower()) != None:
        PrintStatusMessage('Requesting to add admin')
        CheckIfAdmin(CurrentSMS)
    if re.match('delete admin ',currentSMSText.lower()) != None:
        PrintStatusMessage('Requesting to delete admin')
        CheckIfAdmin(CurrentSMS)
    if re.match('add reseller ',currentSMSText.lower()) != None:
        PrintStatusMessage('Requesting to add reseller')
        CheckIfAdmin(CurrentSMS)
    if re.match('delete reseller ',currentSMSText.lower()) != None:
        PrintStatusMessage('Requesting to delete reseller')
        CheckIfAdmin(CurrentSMS)
    if re.match('all staffers',currentSMSText.lower()) != None:
        PrintStatusMessage('Requesting to find all system resellers')
        CheckIfAdmin(CurrentSMS)
```

### Appendix 2: The String module

The **String** module is used to sanitise the response from the Fodder, to take out any characters that could cause trouble when logging the transaction in the PostgreSQL database.

```
@classmethod
def SanitizeResponse(cls,CurrentSMS,Response):
    # Take out any characters that could cause trouble to i

    cls.PrintStatusMessage('Sanitizing response from Fodder

    Response = string.replace(Response,'"','')
    Response = string.replace(Response,',','')

    cls.PrintStatusMessage('Response sanitized')

    cls.LogRecord(CurrentSMS,Response,'Y')
```

### Appendix 3: The math module

The math module is used compute how many blocks a message need to be break into if it exceeds 160 characters. This part of the code Oliver developed it.

```
def SendResponseSMS(CurrentSMS,Response):
    # This is the function that sends a response back to the user

    currentNumber = CurrentSMS[1]

    # Sometimes the answer is longer than 160 characters, so we have to break it up into several messages.
    individualMessageLength = 155 # max length of an individual message
    responseLength = len(Response)
    # how many blocks does the message need to be broken into
    responseBlockCount = int(math.ceil(float((responseLength))/float(individualMessageLength)))

    for i in range(1,responseBlockCount+1): # loop over blocks
        blockStart = (i-1)*individualMessageLength
        blockEnd = i*individualMessageLength
        if i == responseLength: # for the last block, we just send the remainder of the string
            blockText = Response[blockStart:]
        else: # for all the other blocks, we send a specific number of characters, followed by [continued]
            blockText = Response[blockStart:blockEnd]

        if (responseBlockCount >= 2):
            blockCounter = '(%s/%s)' % (i,responseBlockCount)
            blockText = blockCounter + blockText

        SendSMSSerial(currentNumber,blockText)

    LogRecord(CurrentSMS,Response,'Y')
```

### Appendix 4: The pyodbc, time, and serial module

The *pyodbc module* is what is used to connect to the internal PostgreSQL database and the Burro Server. The *time module* is what is used to check the GSM module for new message after every five (5) seconds. The *serial module* is what is used to connect to the GSM module. This part of the code was developed by Oliver.

```

### INITIATE THE PROGRAM -----
marathonConnection = pyodbc.connect(DSN='marathonscript-connector') # Connection to the local postgres database as
fodderConnection = pyodbc.connect('DRIVER=FreeTDS;SERVER=10.11.5.2;PORT=1433;DATABASE=Fodder2.1Live;UID=%s;PWD=%s;T
serialConnection = serial.Serial("/dev/serial0", baudrate=115200, timeout=3.0) # Connection to the serial0 port, wh
serialConnection.write('\r\n') # Clean the serial terminal in case some data is left
time.sleep(1)
serialConnection.write('AT+CMGF=1\r\n') # Set the SIM card to SMS mode and tell it that messages will be passed as
time.sleep(1)

### RUN INFINITE LOOP -----
while True:

    # Query Fona's inbox and return a string with the query response
    PrintStatusMessage('Querying Fona\'s inbox')

    serialConnection.write('\rAT+CMGL="ALL"\r')
    time.sleep(2)

    responseString = ''

    while serialConnection.inWaiting() > 0:
        responseString += serialConnection.read(1)

    # The following line contains a test string for if you temporarily cannot access the modem
    # responseString = '\n+CMGL: 11,"REC READ","57p666131323","", "16/11/18,18:00:51+00"\nTest 2\r\n'

    # This function initiates a series of functions to treat the text message
    ReadFonaResponse(responseString)

    PrintStatusMessage('waiting...')
    time.sleep(5)

```

## Appendix 5: The user manual

### 1. Help functionality to tell the users how to structure their Marathon commands and Fodder queries

All Marathon users can use this functionality. It returns the list of queries the system can process and their structures. The queries to get help with Marathon are:

*“help” this will return the two help categories:*

*Help with marathon commands and help with fodder queries*

*“help with marathon commands” will return all the Marathon commands*

*“help with fodder queries” will return all the fodder queries*

## 2. Add user:

Marathon is designed for Ghanaian numbers, therefore, to successfully add a user, make sure you start entering the user's number from the **zero**. Only Marathon Administrators can add a user. Query structures to add any user:

add admin *[name]* with number *[number]*

add staffer *[name]* with number *[number]*

add reseller *[name]* with number *[number]*

## 3. Deleting user

Only Marathon Administrators can delete any user. Make sure to start the user's number from the

**zero**. The queries to delete any user type are:

delete admin *[number]*

delete staffer *[number]*

delete reseller *[number]*

## 4. Find Marathon users

Only Marathon Administrators and Marathon Staffers can find a Marathon user. Marathon Administrators can find all the Marathon Staffers and as well as Marathon Resellers, but Staffers can only find Marathon Resellers. The queries to find Marathon users are:

all staffers

*staffers in [region]*

*staffer named [name of person]*

all resellers

*resellers in [region]*

*reseller named [name of person]*

## **5. The ten (10) different Fodder questions. Marathon Resellers are limited to the first four (4):**

### **1.What is my Reseller ID?**

Every marathon user can execute this query, the search is based on reseller's phone number as this will give a specific result indicating the reseller's name and ID number, which is then used for subsequent queries.

Query structure: reseller *[phone number of reseller]*

### **2. What is my transaction history?**

This query answers the following questions:

- How much does she owe us or balance on the account?
- When did she last pay us, and how much, with receipt no?

statement *[ID of reseller]*

### **3. What is my sales status?**

This query answers the following questions:

- Is she good, better, or best status?
- What is her paid T90?

sale status *[ID of reseller]*

### **4. What is the price for product X?**

This query answers the following questions:

- What is the SRP a product?
- What are the good, better, and best prices for the product?

price *[product name]*

### **5. What is reseller X details?**

This query answers the following questions:

- What is her contact details?
- Where is she located?

details *[product name]*



## 6. What is reseller X Job card?

This query provides answers to the following questions

- Does she have any faulty products?
- Have they been fixed?
- Where are they located?

Job card [ID of reseller]

## 7. How many of product X are in the stock?

Getting total from all stock locations: count [*product name*]

Getting total from specific stock locations: count [*product name, stock location name*]

## 8. What is the latest note on her account?

Last Note [*ID of reseller*]

## 9. Who is the last person who contacted her?

Who called [*ID of reseller*]

## 10. What was the last product that was returned by the reseller?

Last returned [*ID of reseller*]

## Appendix 6: Oliver's Hardware Manual

### Hardware overview:

The system includes the following hardware. Some of them might be stored inside the case during transportation.

- Raspberry Pi 3.0 preset with a micro-SD card with all the necessary software
- White Case (3D Printed)
- Adafruit GSM Text Module
- Lithium-ion battery
- Flexible GSM antenna
- Dual-output USB charger
- USB cables (2)
- Assorted extra jumper cables

### Sim Card

The system requires a standard-size sim card from a GSM compatible carrier like MTN.

## Setting Up the Hardware

Most of the system should already be assembled, but some parts must be added before it can be turned on. Before going through this guide, please make sure you have the following things ready:

- A standard-size sim card from a GSM compatible carrier like MTN
- An Ethernet cable for connecting the system to Burro's network

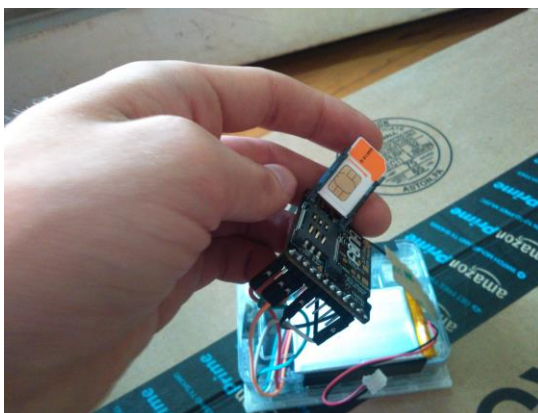
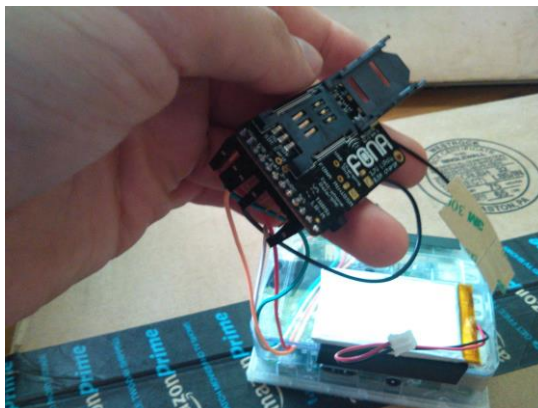
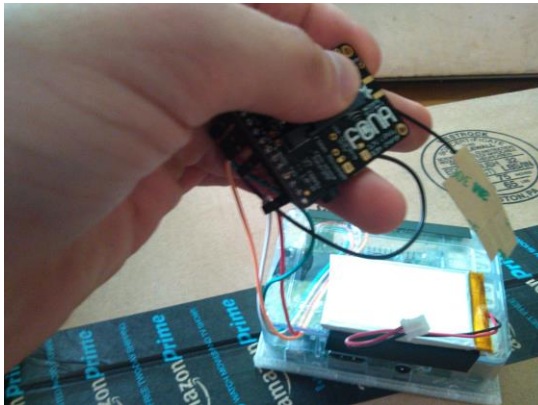
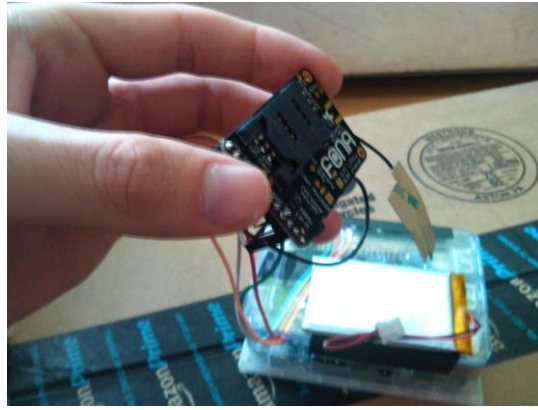
Before starting, please read over the guide and let me know if you have any questions.

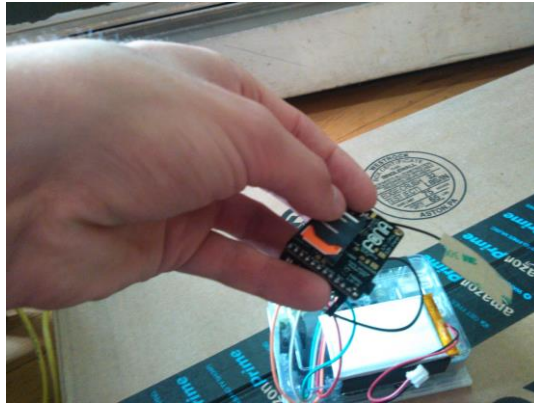
The following are the steps required to set up the Marathon system:

1. Gently lift the top off the case

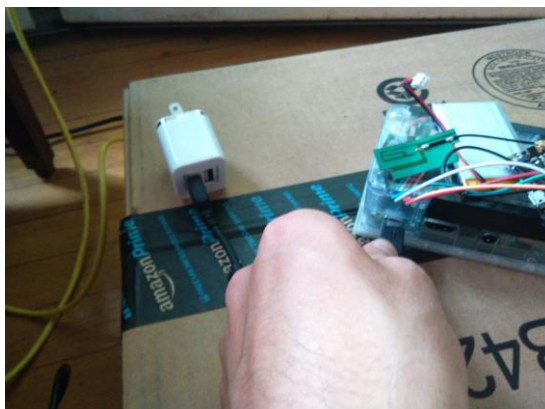
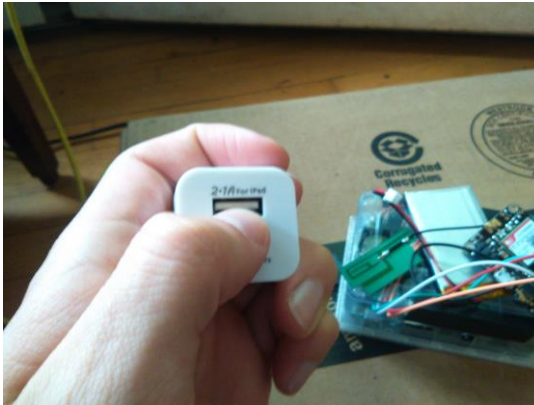


2. Remove any bubble wrap that might be sitting inside the lid of the case to protect the system during transport
3. Gently lift the GSM module and insert the sim card



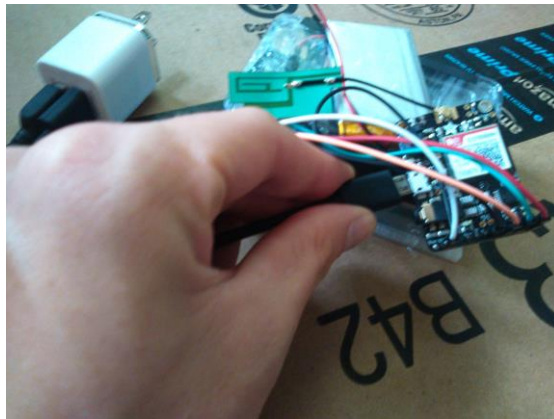
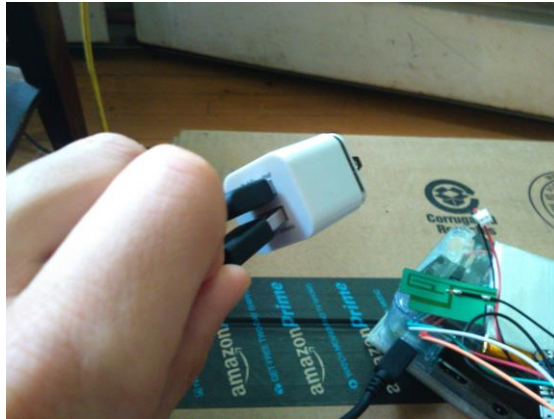


4. Connect the 2-1A for iPad USB socket to the Raspberry Pi Micro-USB power port

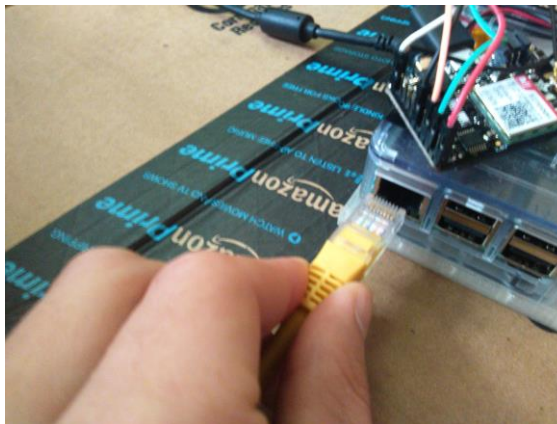


5. Connect the 1-0A Q (???) to the Adafruit Fona micro-USB power port





6. Connect the Ethernet cable from your router to the Raspberry Pi

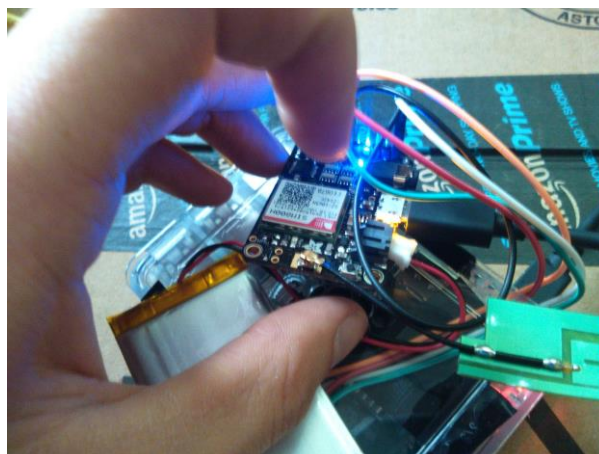
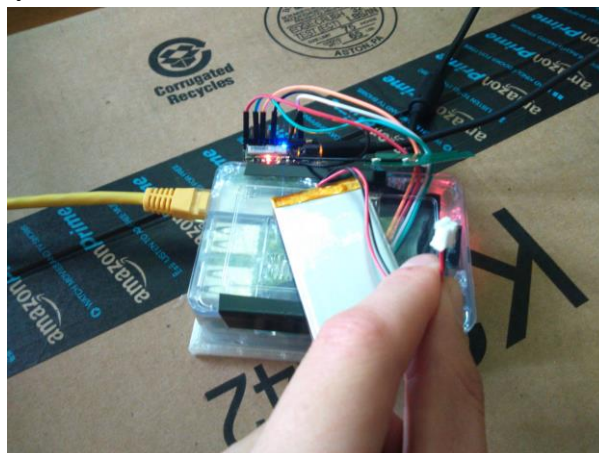


7. Plug in the USB power supply



8. The system should power on automatically. At this point, I will get an email notifying me that the system has been powered on and sending me all the IP address it has been assigned so I can use my username and password to SSH into it. The blinking pattern on the machine should look like this: <https://youtu.be/-uLSmFpGEjs>

9. Attach the battery to the Adafruit Fona Unit

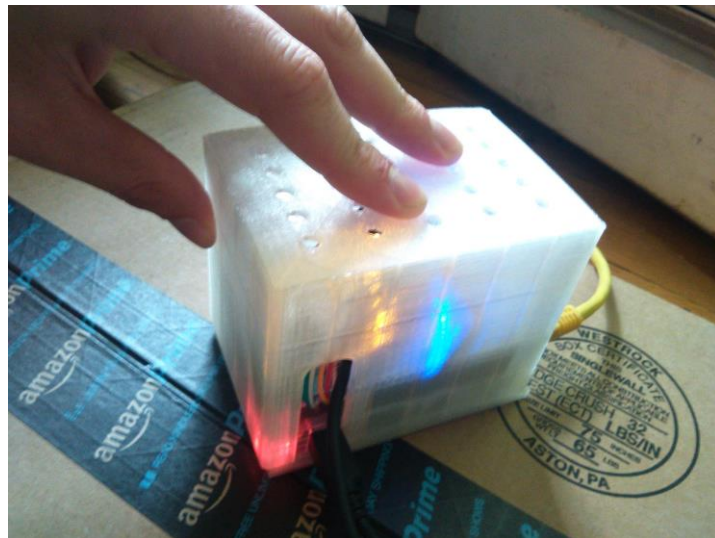


On that picture, the cable is not pushed in. It should be pushed all the way in as you can see in this video that also shows the new blinking pattern:

<https://youtu.be/ZM1aKjEBYiE>

When sending text messages, the Fona GSM Modem has a momentary increase in power demand. Instead of an expensive current controller, the system just uses a regular cell phone battery to assist the USB power for a few seconds at a time when texts need to be sent. Therefore, the battery needs to be attached. The battery needs to be attached after the unit turns on so it can register it as being there. If the unit is restarted, you might need to unplug the battery, turn off the unit (by pulling the plug), turn it back on again and then plug in the battery.

10. Close the lid while gently fitting all the loose cables inside it



11. Send me an email, letting me know you've set up the kit.

## **Appendix 7: Oliver's Software Manual**

### **Raspbian/NOOBS**

The system was developed to run on the Linux distribution NOOBS 2.1, based on Raspbian. For security purposes, it is crucial that you schedule automated updates to ensure that you always have the latest and most secure version of Raspbian. A guide on how to use Crontab to schedule automatic updates can be found here

### **Python**

The script requires Python 2.7 with the Anaconda package.

### **SSH Connection**

If an SSH connection is needed, a guide on how to install the necessary software and generate a key can be found here.

### **Freeing the Serial Connection**

On newer versions of Raspbian, the Serial0 port is per default reserved for the system. To communicate with the Fona modem, you must take steps to free up the port. A guide on these steps can be found here.

### **Postgres Database**

Marathon uses a small, locally hosted Postgres database to keep track of trusted numbers and to store a log of all communication through the system. This database works as a buffer to make sure no numbers that are not already trusted can communicate with Fodder.

To install Postgres, run the following command:

```
sudo apt-get install PostgreSQL
```

It is recommended to use Crontab to schedule automatic updates of PostgreSQL to keep the database secure.

To grant root access to the database, enter the following command in the terminal:

```
sudo -u Postgres createuser -s root
```



Now, to create the database write the following command to the terminal:

```
sudo createdb Marathon
```

Now you can enter the database through psql (a front-end installed with postgresql) by writing:

```
sudo psql Marathon
```

The code to create the table to hold the trusted numbers is:

```
create table TrustedNumbers (  
    ID serial primary key,  
    Number varchar (30) NOT NULL,  
    Name varchar (60) NOT NULL  
);
```

The code to create the table with communication records is:

```
create table Records (  
    ID serial primary key,  
    Number varchar (30) NOT NULL,  
    Message varchar (160),  
    Response varchar (200),  
    Trusted varchar (1)  
);
```

Now you will need to create a user for the script with the necessary privileges to read and write to the tables. This is done by entering the following code in psql:

```
create user marathonscript with password  
'YAfLTsq0tY61NrUwI9mYu5CBrt8lAkY';  
  
grant all privileges on TrustedNumbers to MarathonScript;  
  
grant all privileges on Records to marathonscript;  
  
grant usage, select on sequence records_id_seq to marathonscript;
```

You can quit psql with the following command:

```
\q
```

Please note that this just sets up the back-end structure for the database. You will still need to install and configure drivers before you can access the database through Python.

## **ODBC and pyODBC Database Connections**

The next step is to install the drivers necessary to connect to databases through SQL. To be able to use the same protocol for both the internal Postgres database and Burro's MSSQL hosted database, Marathon uses pyODBC to create ODBC-based connections based on drivers and connection protocols stored externally to the script.

pyODBC has several dependencies which you can install by running the following in terminal:

```
sudo apt-get install python-dev  
sudo apt-get install unixodbc  
sudo apt-get install unixodbc-dev
```

Now, to install pyodbc, enter the following code into the terminal:

```
sudo pip install pyodbc
```

### **Database Drivers**

This the previous steps set up the ODBC connection framework as well as pyODBC, but you still need to configure drivers. For the postgres database, Marathon uses the native postgresSQL driver. For connection to Burro's MSSQL hosted database Fodder, it uses freeTDS.

To install the PostgreSQL drivers, enter the following into the terminal:

```
sudo apt-get install odbc-postgresql
```

To install the freeTDS drivers, write

```
sudo apt-get install tdsodbc  
sudo apt-get install freetds-dev freetds-bin unixodbc-dev tdsodbc
```

Note, you are about to edit root files in Linux. This will require that you open them in sudo mode, for example by using sudo "nano <filename>" in the terminal.

Now, go check the /etc/odbcinst.ini file to make sure it reads something like the following:

```
[PostgreSQL ANSI]  
Description=PostgreSQL ODBC driver (ANSI version)  
Driver=psqlodbc.so  
Setup=libodbcpsqlS.so
```

```
Debug=0
CommLog=1
UsageCount=1
```

```
[PostgreSQL Unicode]
Description=PostgreSQL ODBC driver (Unicode version)
Driver=psqlodbcw.so
Setup=libodbcpsqlS.so
Debug=0
CommLog=1
UsageCount=1
```

```
[FreeTDS]
Description=FreeTDS Driver
Driver=/usr/lib/arm-linux-gnueabi/odbc/libtdsodbc.so
Setup=/usr/lib/arm-linux-gnueabi/odbc/libtdsS.so
```

If any of this is missing, just copy it in

Now, set up the connections to both Marathon's internal SQL server and Burro's MSSQL server. To do this, edit the /etc/odbc.ini file to make sure it includes the following lines (hidden passwords and information is written as <Description>):

```
[marathonscript-connector]
Description      = PostgreSQL connection to Marathon database
Driver           = PostgreSQL Unicode
Database         = Marathon
Servername       = localhost
Username         = <Username>
Password         = <Password>
Port             = 5432
Protocol         = 9.3
ReadOnly         = No
RowVersioning    = No
ShowSystemTables = No
ShowOidColumn    = No
FakeOidIndex     = No
```

```
ConnSettings      =  
[fodder-connector]  
Driver = FreeTDS  
Server = <Local Database IP>  
Port = 1433  
Database = <Database Name>  
TDS_Version = 8.0
```

Note, the /etc/odbc.ini file defines the database connections and refers to driver definition stored in /etc/odbcinst.ini.

## **Crontab**

The script is set up to run automatically in the background on boot. This is done using

Crontab. You can find a description of Crontab at this link:

<https://www.raspberrypi.org/documentation/linux/usage/cron.md>

To set up the Crontab task on the pi, enter the following into the terminal:

```
sudo crontab -e
```

This opens the Crontab document. Edit it to include the following line:

```
@reboot python /home/marathon/Marathon.py &
```

## References

- Joshi, R., & Pathak, L. (2014). *A survey of SMS based Information Systems*. Retrieved 6 October from <https://arxiv.org/ftp/arxiv/papers/1505/1505.06537.pdf>
- Moller, O. (2016). *Marathon Hardware Setup Manual*. Unpublished manuscript, The School of Engineering, *Brown University*, Providence, Rhode Island, United States.
- Moller, O. (2016). *Marathon Software Setup Manual*. Unpublished manuscript, The School of Engineering, *Brown University*, Providence, Rhode Island, United States.