CE903 Group Project: Lightweight Cluster Monitor

Ercan Ezin , Kyung H. Cho , Matt Cockman, Oluwadunsin S. Dabor, Sara B. Alharbi

Contents

[Introduction 3](#_Toc477811341)

[System Design 4](#_Toc477811342)

[3. Implementation 5](#_Toc477811343)

[3.1 CPU and Memory 5](#_Toc477811344)

[3.2 UDP Sender program 9](#_Toc477811345)

[3.3 UDP Listener program 9](#_Toc477811346)

[3.3.1 UDPlistener.py 9](#_Toc477811347)

[3.3. 3 Technical Issues 11](#_Toc477811348)

[3.3. 4 Development 11](#_Toc477811349)

[3.5 Web Application 12](#_Toc477811350)

[3.5.1. Implementation 12](#_Toc477811351)

[3.5.3. Conclusions 17](#_Toc477811352)

[3.6. Project Management 18](#_Toc477811353)

[4. Testing 21](#_Toc477811354)

[4.1. UDP Sender 21](#_Toc477811355)

[4.1. UDP Listener 21](#_Toc477811356)

[4.2. Web Application 24](#_Toc477811357)

[5. References 26](#_Toc477811358)

# Table of Figures

[Figure 1 - Organisation of Brambles 3](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840014)

[Figure 2 - Use Case Diagram showing user interaction with system 5](#_Toc477840015)

[Figure 3 - Data Flow Diagram 6](#_Toc477840016)

[Figure 4 - Sequence Diagram 6](#_Toc477840017)

[Figure 5 - Linux /proc/stat file example 7](#_Toc477840018)

[Figure 6 - Calculating CPU usage 8](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840019)

[Figure 7 - Linux /proc/meminfo Example 9](#_Toc477840020)

[Figure 8 - Calculating Memory Usage 10](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840021)

[Figure 9 - Code showing the sending of a UDP message 12](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840022)

[Figure 10 - Structure of 'data\_dic' 13](#_Toc477840023)

[Figure 11 - Bootstrap Grid System Example 17](#_Toc477840024)

[Figure 12 - A screenshot from GUI screen 18](#_Toc477840025)

[Figure 13 - Pop up containing update time 18](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840026)

[Figure 14 - An example refresh loop of the index page 19](#_Toc477840027)

[Figure 15 - Info button 20](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840028)

[Figure 16 - Manual Page 20](#_Toc477840029)

[Figure 17 - Application Files Tree 21](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840030)

[Figure 18 - Mock data generator 25](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840031)

[Figure 19 - Sprint 1 Board and Tasks 27](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840032)

[Figure 20 - Sprint 2 board with completed tasks 28](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840033)

[Figure 21 - Sprint 3 board with completed and uncompleted tasks 28](file:///M:\Matt\Year%204\CE903\Final%20Report.docx#_Toc477840034)

# 1. Introduction

## **1.1. Problem Domain**

The objective of this project was to build a performance monitoring system for the Raspberry Pi clusters in use by the CSEE department. The system should be as lightweight as possible and minimise performance requirements.

The Raspberry Pi cluster, also known as ‘brambles’, consists of 94 Raspberry Pi computers connected by switches and two gateway machines (cseepiman1 and cseepiman2). The cluster is currently running on the University network. Each Raspberry Pi is a single board computer featuring an ARM processor roughly equivalent of 700 MHz [1] and runs on a Linux operating system.

The Pi nodes are accessible by SSH via two gateway machines as described in figure 1.

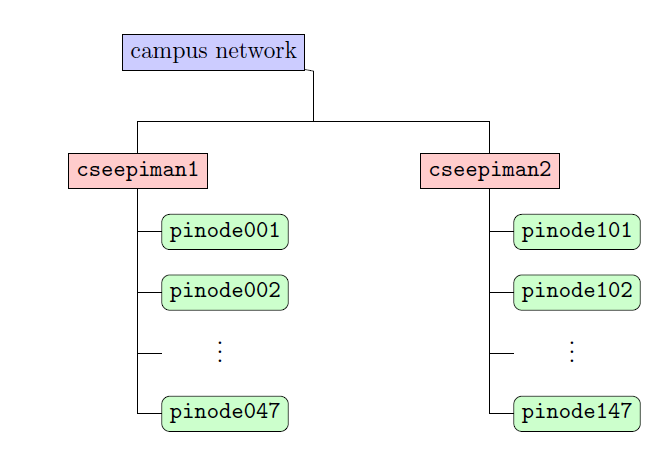


Figure - Organisation of Brambles

The system will run across all of these nodes and report back to a central application allowing the user to view the performance data for each node.

## **1.2. Methodology**

We used agile methodology

## **1.3. Definition of Terms and Abbreviations**

|  |  |
| --- | --- |
| ARM | Advanced RISC Machine, is a family of [reduced instruction set computing](https://en.wikipedia.org/wiki/Reduced_instruction_set_computing) (RISC) [architectures](https://en.wikipedia.org/wiki/Instruction_set) for [computer processors](https://en.wikipedia.org/wiki/Central_processing_unit). |
| Brambles | A cluster of 94 Raspberry Pi systems running on the University Network |
| Daemon | This is a computer program that runs as a background process. Mostly used on Unix operating systems. |
| Gateway | A [computer](https://en.wikipedia.org/wiki/Computer) configured to perform the tasks of a gateway and provides system interoperability |
| GUI | Graphical User Interface |
| JSON | JavaScript Object Notation is an open-standard and language independent format that uses human-readable text to transmit data objects. |
| Node | This refers to devices (Computers, printers) /points on a larger network |
| Raspberry pi | A single-board computer of the size of credit card which uses ARM processor (equivalent to a 700MHz Pentium), and has a built in 100Mb/s Ethernet |
| SSH | Secure Shell (SSH) is a [cryptographic](https://en.wikipedia.org/wiki/Cryptography) [network protocol](https://en.wikipedia.org/wiki/Network_protocol) for operating network services securely over an unsecured network |
| TCP | Transmission Control Protocol. It provides a reliable communication and delivery of data between applications running on hosts communicating by an IP network |
| Trello | Trello is a [web-based](https://en.wikipedia.org/wiki/Web_application) [project management application](https://en.wikipedia.org/wiki/Project_management_software) |
| UDP | User Datagram Protocol. It uses a simple connectionless transmission model with a minimum protocol mechanism. |

## **1.4. Technologies**

The performance gathering system was written in C. C is the lightest possible language available for this task and being frugal with resources is extremely important. Although this presented some difficulties as none of the team were strong in C these challenges were overcome.

The cluster monitor server was programmed in Python. Python was chosen as it has standard access to networking requirements, strong support for transforming data, and the team have familiarity with it. Additionally the server has no resource constraints so the programming language choice is, to an extent, immaterial.

Networking is done via UDP. UDP is chosen as it is frugal with resources as it uses a ‘fire and forget’ approach. Any packet loss is easily dealt with by the system because losing packets is not a significant problem.

A web application is created via bootstrap, JQuery and JavaScript.

## **1.5. Overview of Document**

# 2. System Design

## 2.1 Summary of Requirements

## 2.2. System Architecture

Figure two shows how a user will interact with the system. As discussed in the requirements specification [1] ‘The user starts the system by running a script that executes a start function on each node. This function launches a daemon that determines the current performance of the node, and, using UDP, sends this information to the server. The performance data is saved as a JSON file, and is displayed on a web application. The data is updated every 10 seconds.’ Clearly the system is organised into several components. These are:

* Calculating CPU and Memory data
* Sending a UDP message
* Receiving a UDP message
* Converting UDP message to JSON format
* Web application

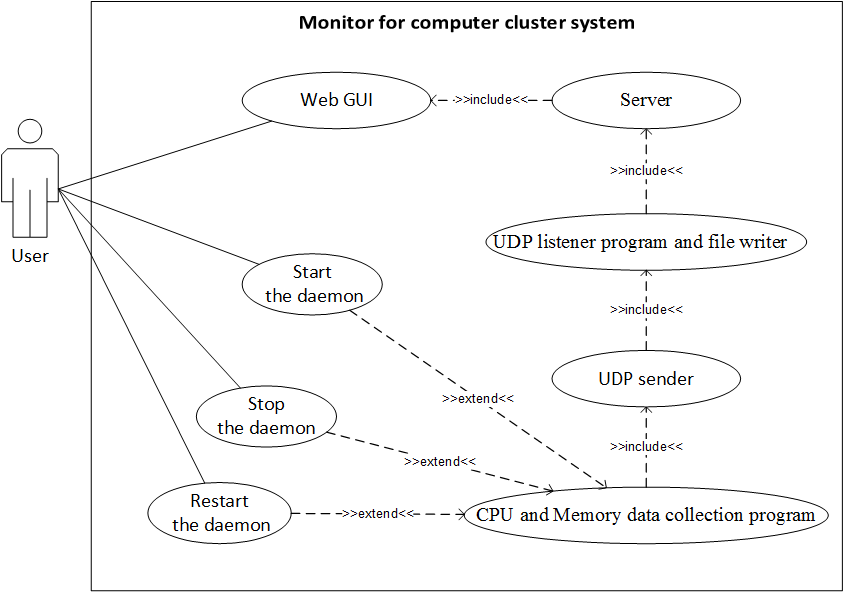


Figure - Use Case Diagram showing user interaction with system

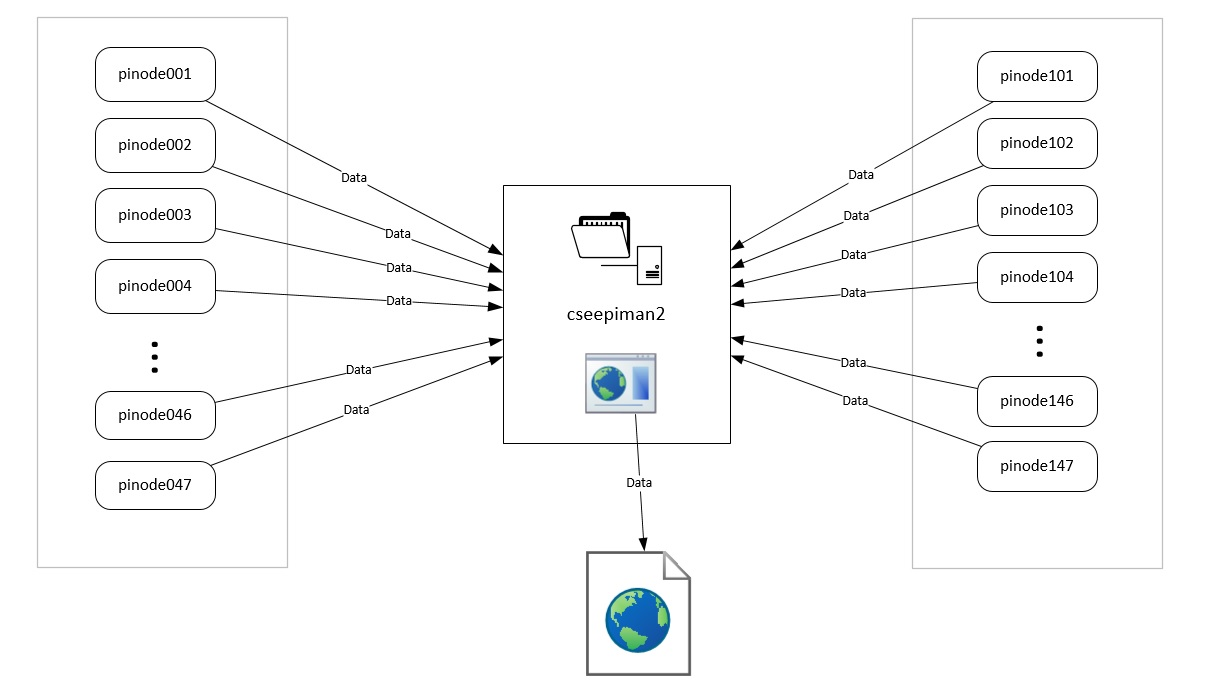


Figure - Data Flow Diagram

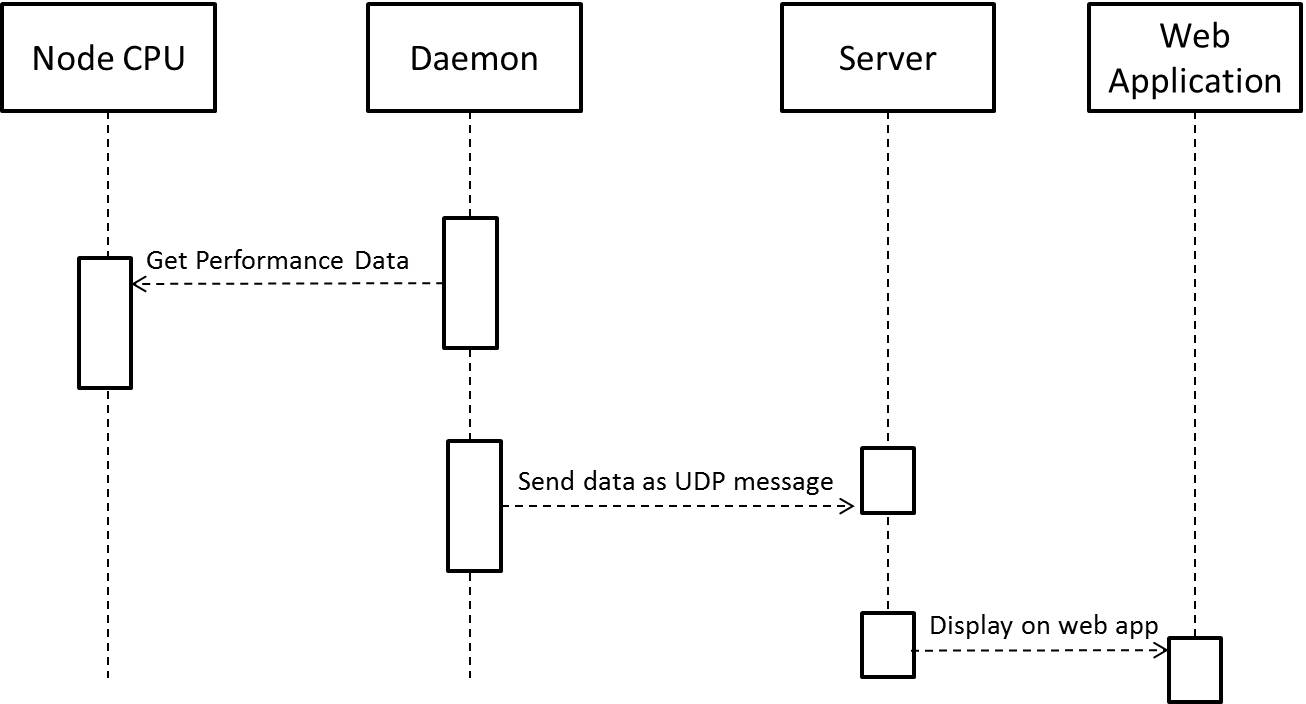


Figure - Sequence Diagram

# 3. Implementation

## 3.1 CPU and Memory

This module of the application as defined in the SRS document is designed to retrieve the CPU and memory utilization metrics on each Raspberry Pi node every millisecond. The function was developed using C (requirement for a light weight application). An average is taken of the CPU and Memory every second and sent to the UDP Sender function.

Raspberry Pi nodes run on the Linux operating system. The Linux operating system has a virtual file system called “/proc” that stores runtime system information. The /proc directory also contains information about system hardware and any current processes running on the system. The information from this file was used to determine the CPU usage and Memory usage.

The content of the /proc/stat directory was used in calculating the CPU usage level. Various pieces of information about kernel/system activities are stored within the /proc/stat file when a system is booted. Information in the /proc/stat file varies for different architecture. The /proc/stat file information for the version of the Linux running on the Raspberry Pi node is as shown in the figure below.

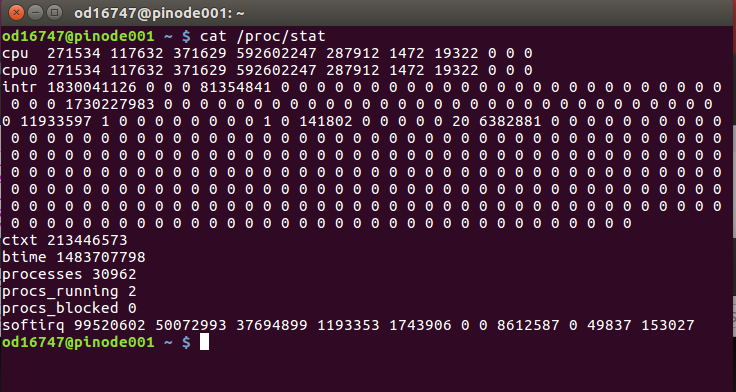


Figure - Linux /proc/stat file example

The first line ‘cpu’ gives the total values for all the other ‘cpuN’s lines (N >=0) running on the system at the particular time. Each Raspberry pinode has only one cpu, so the first line ‘cpu’ and the second line ‘cpu0’has the same value. Each of the columns specify the time that CPU has spent performing different jobs.

* First Column: **user** = normal processes executing in the user mode
* Second Column: **nice =** niced processes executing on the user mode
* Third Column: **system** = processes executing in the kernel mode
* Fourth Column: **idle** = waiting
* Fifth Column: **iowait** = waiting for I/O to complete
* Sixth Column: **irq =** servicing interrupts
* Seventh Column: **softirq =** servicing soft interrupts

To calculate the CPU usage; two readings at specified intervals (reading 1 and reading 2) are taken and calculated as follows:

* Total\_reading1= (user\_1 + nice\_1 + system\_1 + idle\_1 + iowat\_1 + irq\_1 + softirq\_1)
* Used\_reading1= (user\_1 + nice\_1 + system\_1 + iowat\_1 + irq\_1 + softirq\_1)
* Total\_reading2 = (user\_2 + nice\_2 + system\_2 + idle\_2 + iowat\_2 + irq\_2 + softirq\_2)
* used\_reading2= (user\_2 + nice\_2 + system\_2 + iowat\_2 + irq\_2 + softirq\_2)
* CPU Usage= ((used\_reading2-used\_reading1) / (Total\_reading2- Total\_reading1))\*100

The c-program function below reads the values from the virtual file and calculates the CPU usage

Figure - Calculating CPU usage

while (i<10)

{

//CPU FIRST READING

prostf = fopen("/proc/stat", "r");

fscanf(prostf, "%s %Lf %Lf %Lf %Lf %Lf %Lf %Lf", c, &a[0], &a[1], &a[2], &a[3], &a[4], &a[5], &a[6]);

fclose(prostf);

usleep(100000);

//CPU SECOND READING

prostf = fopen("/proc/stat", "r");

fscanf(prostf, "%s %Lf %Lf %Lf %Lf %Lf %Lf %Lf", c, &b[0], &b[1], &b[2], &b[3], &b[4], &b[5], &b[6]);

fclose(prostf);

loadavg = (b[0] + b[1] + b[2] + b[4] + b[5] + b[6])-(a[0] + a[1] + a[2] + a[4] + a[5] +a[6]);

total= (b[0] + b[1] + b[2] + b[3] + b[4] + b[5] + b[6])-(a[0] + a[1] + a[2] + a[3] + a[4] + a[5] +a[6]);

cpuUsage=(loadavg/total) \* 100;

printf("The current CPU usage is : %Lf\n", cpuUsage);

newCpu+=cpuUsage;

usleep(100000);

i++;

}

//cpu usage for 1sec

newCpu= newCpu/10;

The /proc/meminfo file provides valuable information about a systems RAM usage. The information from this file was used to calculate the memory usage at every milliseconds and the average calculated over a period of 1 second. The figure below show a sample of the proc/meminfo content from a Raspberry pinode

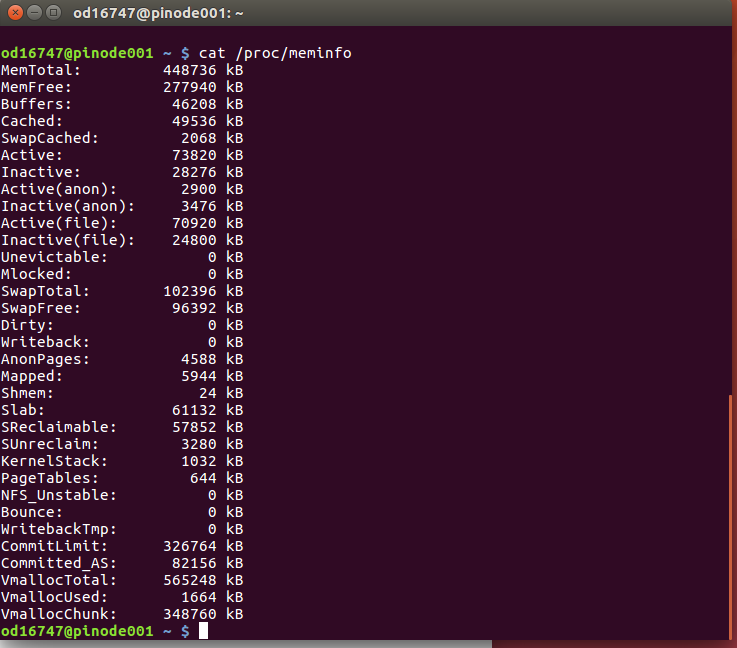


Figure - Linux /proc/meminfo Example

To calculate the amount of memory used at a particular time the formula below was used:

Memory Usage = ((Memtotal – (MemFree + Buffer + Cached)) / MemTotal) \*100

The program segment below (figure 4) was used in determining the memory usage calculation

Figure - Calculating Memory Usage

nt parseLine(char\* line)

{

//This assumes that a digit will be found and the line ends in "Kb".

//this function removes the KB at the end of the value

int i = strlen(line);

const char\* p = line;

while (\*p <'0' || \*p > '9') p++;

line[i-3] = '\0';

i = atoi(p);

return i;

}

While(i<10)

prostf = fopen("/proc/meminfo", "r");

while (fgets(buff, 128, prostf)!= NULL){

if (strncmp(buff, "MemTotal:", 7) == 0){

memFirst[0] = parseLine(buff)/1024;

}

if (strncmp(buff, "MemFree:", 7) == 0){

memFirst[1] = parseLine(buff)/1024;

}

if (strncmp(buff, "Buffers:", 7) == 0){

memFirst[2] = parseLine(buff)/1024;

}

if (strncmp(buff, "Cached:", 3) == 0){

memFirst[3] = parseLine(buff)/1024;

break;

}

}

fclose(prostf);

loadavg = (memFirst[0]) - (memFirst[1]+memFirst[2]+memFirst[3]);

total= (memFirst[0]);

memoryUsage=(loadavg/total) \* 100;

newMemory+=memoryUsage;

usleep(1000000);

i++;

}

newMemory=newMemory/10;

## 3.2 UDP Sender program

### 3. Implementation

#### Sending UDP Packets

This component is responsible for formatting the processed performance data, and sending it as a UDP message to a separate listener component. This component has the following responsibilities:

* Reformat the performance data into string format
* Obtain and format the current time and date
* Obtain current node name
* Parse all data into one message
* Set up UDP socket
* Send UDP message

#### Technologies

This component was programmed in C. This bought up a few issues mainly due to lack of experience. C is very strict with memory allocation, and additionally C uses pointers. This required some study on how these features worked. C usually requires the exact size of a variable to be defined. This means that a lot more thought needs to go into the space complexity of a program.

As a group we decided to use UDP to send the message. This was because UDP is a lightweight messaging protocol. There is a danger of occasionally dropping packets but that is not critical for this system. A UDP message in C is sent through a socket (see next section) with the UDP protocol specified in the socket creation.

#### Sockets in C

A socket is used to send the performance data. A socket is essentially a point for sending or receiving data (in this case sending). There are two main parts to using a socket in C:

1. Setting up the socket
2. Sending a message

A socket is set up first by using the socket() method. This method is where the UDP protocol is set (see the next section for an example). Then a struct is set up to provide information about the address (such as the port and IP address).

The second stage is to actually send a message. This is done via the sendto() method. This method takes the following parameters as arguments:

* int socket
  + The socket created via socket()
* const void \*message
  + The message to send
* size\_t length
  + The length of the message to send
* int flags
  + This was set to 0 for our project
* const struct sockaddr \*dest\_addr
  + The destination address
* socklen\_t dest\_len
  + The length of the sockaddr structure pointed to by the *dest\_addr* argument

Figure - Code showing the sending of a UDP message

int main(int argc, char\*\*argv){

char data[80];

int handle = socket(AF\_INET, SOCK\_DGRAM, IPPROTO\_UDP );

struct sockaddr\_in addr;

addr.sin\_family = AF\_INET;

addr.sin\_port = htons(5005);

addr.sin\_addr.s\_addr = inet\_addr("192.168.7.200");

memset(addr.sin\_zero, '\0', sizeof addr.sin\_zero);

int addr\_size = sizeof addr;

while(1){

int i=0;

double CPU=0,Memory=0;

while(i<10){

CPU+= getCPU();

Memory+=getMemory() ;

usleep(100000);

i++;

}

CPU = CPU / 10;

Memory = Memory / 10;

//Get Time

time\_t time\_raw\_format;

struct tm \* ptr\_time;

char buffer[50];

time ( &time\_raw\_format );

ptr\_time = localtime ( &time\_raw\_format );

if(strftime(buffer,50,"%H:%M:%S %d.%m.%Y",ptr\_time) == 0){

perror("Couldn't prepare formatted string");

}

char CPUString[100] ;

ToString(CPUString, CPU);

char MemString[100] ;

ToString(MemString, Memory);

char hostname[1024];

gethostname(hostname, sizeof(hostname) -1);

strcpy(data, hostname);

strcat(data, ",");

strcat(data,buffer);

strcat(data,",");

strcat(data, CPUString);

strcat(data, ",");

strcat(data, MemString);

printf("%s\n", data);

printf("Sending Message\n");

int sent\_bytes = sendto(handle, data, strlen(data), 0,

(const struct sockaddr \*)&addr, addr\_size);

}

}

## 3.3 UDP Listener program

The UDP listener program consists of two python programs that are currently deployed on CSEEMAN2 server.

### Implementation

#### 3.3.1 UDPlistener.py

This is a python program that opens a UDP socket, binds to a port (currently set to 50065), and waits for UDP messages.

**UDP\_PORT = 50065sock = socket.socket(socket.AF\_INET, socket.SOCK\_DGRAM)sock.bind(('', UDP\_PORT))**

As messages arrive from Pi nodes, the program stores the data in a python dictionary called ‘data\_dic’.

|  |  |
| --- | --- |
| **Data\_dic Key** | Nodename e.g. pinode001 |
| **Data\_dic Value** | UDP message e.g. pinode001, ,13,4 |

Figure - Structure of 'data\_dic'

**data\_dic={}**

**while True: data, addr = sock.recvfrom(1024) print "received message:", data y = data.strip().split(",") nodename=y[0] data\_dic[nodename]=data print "the following is the content of data\_dic" print data\_dic.values() file\_output(data\_dic,file1)**

The dictionary will always contain the most up-to-date data as the value for each Pi node (dictionary key) gets updated in Memory immediately.

The program outputs the contents of the dictionary into a csv file called ‘cluster\_monitor\_data\_10sec.csv’. This file then gets fed back to DataTransformer.py program.

**## Write the content of dictionary data into .csv filedef file\_output(d, filename): if os.path.exists(file1): os.remove(file1) with open(filename, "a") as out\_file: for k,v in d.items(): line = '{}'.format(v) out\_file.write(line+'\n')**

\*\*\* The full program codes have been attached to Appendix.

#### 3.3.2 DataTransformer.py

This program first uploads the latest data from ‘cluster\_monitor\_data\_10sec.csv’ created by the UDPlistener.py program explained above. It also takes data being used by the web application at the time of processing.

The program compares both data sets and prepare a new csv file, ‘cluster\_monitor\_data.csv’ accordingly following two rules. Function **update\_csv()** implements the task. The rules are:

* For nodes that sent a new CPU & Memory data in the last 1 second, add the new data to the file.
* For nodes that DID NOT send CPU & Memory data in the last 1 second, keep the old data in the file.

**def update\_csv(f1,f2):**

**f\_1=open(f1,"r") result = open(f2, 'a') # open file handle for write for line in f\_1: new\_line=line y = new\_line.strip().split(",") print y[0] pattern=y[0] if y[0]!="Node": found=check(f2,pattern) if found: #print "Node entry exists" else: #print "no data for the node : ", pattern, "Data will be added : ", new\_line result.write(new\_line) result.close() f\_1.close()**

It uses the Python Panda library to reorder the contents of the data by node name in ascending order.

Jsonify Python library was used to convert the csv file (cluster\_monitor\_data.csv) into a json file called ‘cluster\_monitor\_data.json’

**#Transform csv file to a Json fileconvert.jsonify(file2)**

The newly created json file is then picked by the web application to display on the GUI.

**df = pd.read\_csv(file\_tmp)df = df.sort(columns='Node')df.to\_csv(file2, index=False)**

\*\*\* The full program codes have been attached to Appendix.

### 3.3.3 Technical Issues

There was no major technical issue or obstacle that interrupted releases, but designing part of DataTransformer.py took a significant amount of efforts and changes.

Deciding on the suitable file format before converting to Jason file was not a simple task and many considerations had to be put in.

Keeping the dataset consistent and up-to-date while maintaining historic data for nodes failing to send messages was a goal which was rather complicated to achieve.

In the end the program had to be written to step through several data merging and transformation processes.

### 3.3.4 Development Process

**Tools Used**

No specific development IDE was needed as the programs were written in Python language. The codes were written in a plain text editor (gedit) then tested using python 2.7.1.

**Development**

Sprint 1

FR 3.1 was implemented and tested. The first version UDPlistener.py was developed.

**ID: FR 3.1.**

|  |  |
| --- | --- |
| **Title** | Listen and Receive UDP/TCP messages |
| **Description** | The program shall receive and capture UDP/TCP messages sent by each Pi nodes. |

Sprint 2

* FR. 3.3 was implemented.
* The first version of DataTransformer.py was released.
* The second version of UDPlistener.py was released.

**ID: FR 3.3.**

|  |  |
| --- | --- |
| **Title** | Storing and Maintaining data |
| **Description** | The program shall store the data generated in FR 3.2 in a single file (e.g. txt, xml or json format) as necessary. If a message fails to arrive from a certain node, the program should still keep the old data in the file. |

Improvement into FR.3.1

Data storing was changed from file based to Python dictionary.

Sprint 3

* FR 3.2 was implemented and tested.
* The second version of DataTransformer.py was released.

**ID: FR 3.2.**

|  |  |
| --- | --- |
| **Title** | Data conversion & consolidation |
| **Description** | The program shall convert and consolidate the messages received from Pi nodes in FR 3.1 to a suitable data format as necessary. |

## 3.4. Web Application

### 3.5.1. Implementation

#### Problems Faced:

We decided to use Bootstrap with JQuery for developing the GUI. JavaScript was also used which allowed us to manipulate the data in the cells. Each time the JSON file was read from the local directory, previous tables showing the old data were purged and new tables were then created. This was a problem for us because this as creating a flickering effect which can be annoying for time period changes. However, the customer preferred to have a flickering effect as it indicated new data.

Another problem that we needed to fix is that the date format being sent from nodes may change e.g.: 12/03/2016 or 12.06.2016, and we needed to use a flexible date formatter to handle all kind of incoming formats. After some searches we came across a solution on the internet [2].

#### Key Components:

Bootstrap’s grid system uses a series of containers, rows, and columns to layout and align content. It’s built with flexbox and is fully responsive [1]. This grid system is used to create a neat responsive structure for tables and cells.



Figure - Bootstrap Grid System Example

After building the structure of the main page we implemented the auto-refresh part so that data can be refreshed regularly. We used a JQuery function to call a refresh function within a timed loop, which is 3000 for an optimal case. This refresh function deletes previous created tables, then reads a local JSON data file, and formats the data it read. Finally, a loop creates new HTML elements in a string which is then appended by JQuery to replace the previously deleted html elements. Additionally, as requested by customer and indicated in FR 1, we implemented a pop-up dialog to show latest updated time of the data.

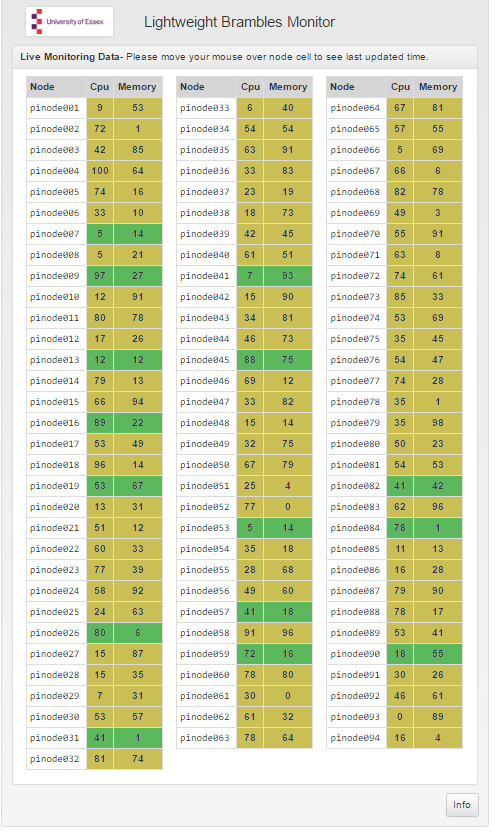


Figure - A screenshot from GUI screen

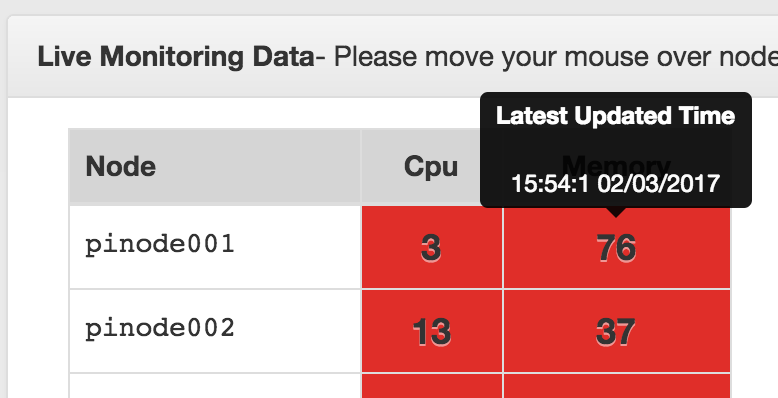


Figure - Pop up containing update time

A basic data flow on GUI starts by running a command while in the index file directory, and then creating a basic python “*http.server*” serving on port 8000 from CSEEPIMAN machines. The GUI part was designed to assume that a JSON file containing updated node info available is being updated regularly.

The first time a user requests the website refresh function calls start looping. Then after the set time, the updated JSON file is read. The colour in each cell represents the freshness of the data. Green colour indicates that data in the cell has been refreshed in the past 10 minutes. Yellow cells indicate that data hasn’t been updated in the last 24 hours. Finally, the red cells indicate that data hasn’t been updated for more than 24 hours.

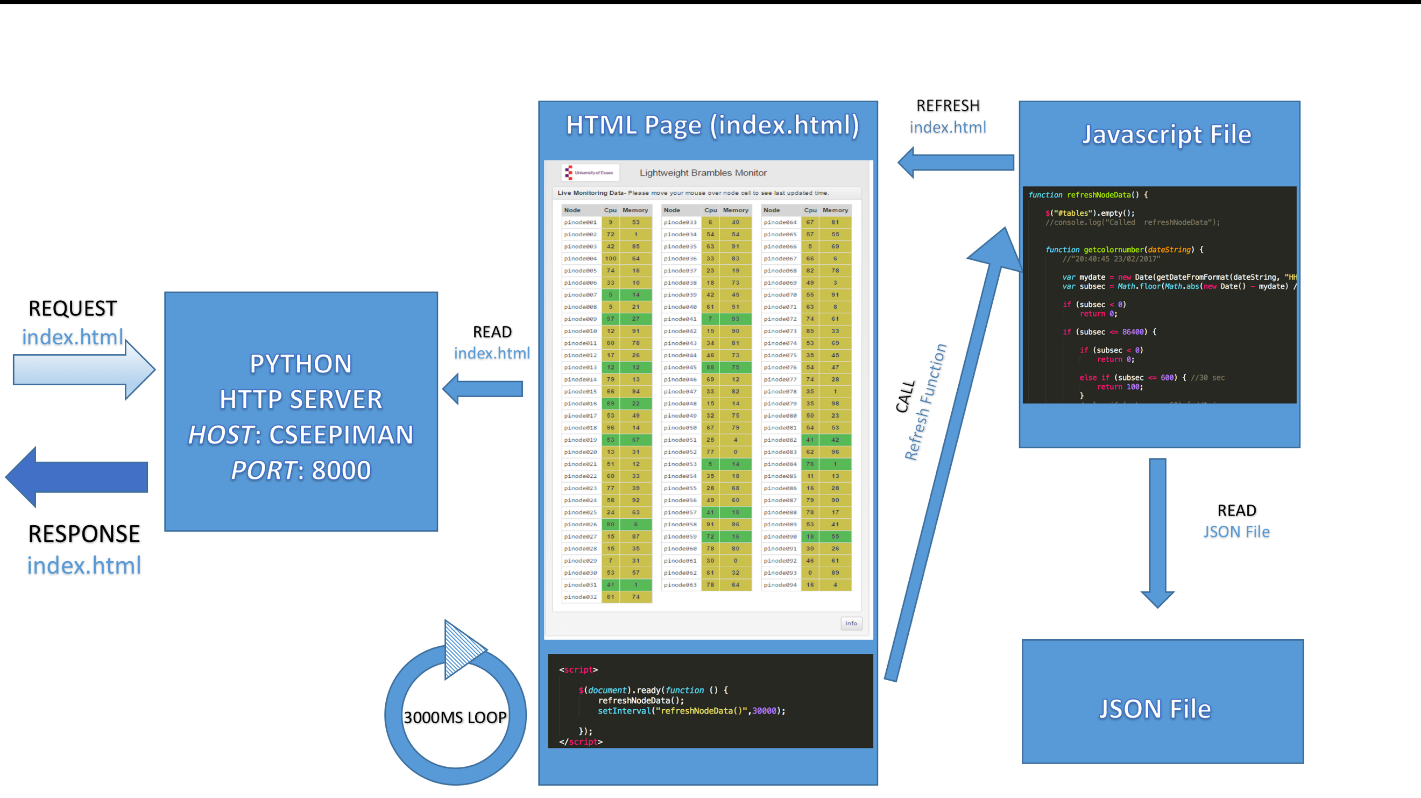


Figure - An example refresh loop of the index page

Additionally, as requested by the customer, we also added a manual page to the codes which can be accessed by clicking on an info button located at the bottom of the generated cells. This location is also then used to redirect user back to index page on info page.

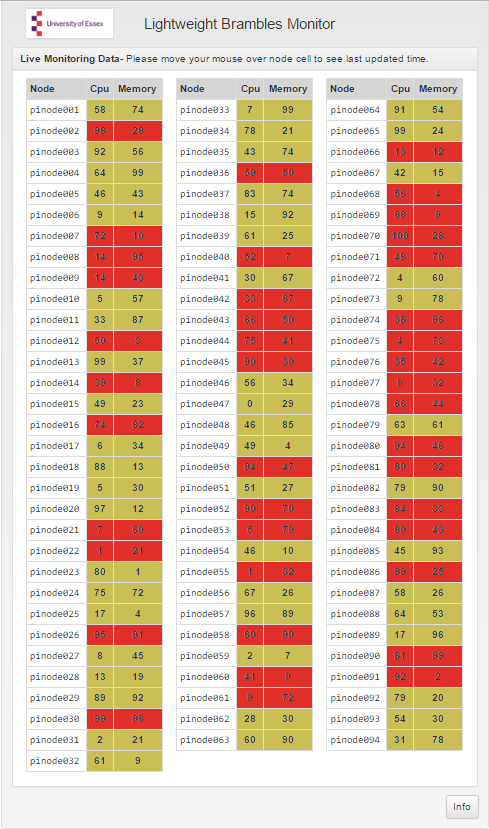


Figure - Info button

The Info button leads to a manual page which we built by exploiting the structure used in the index page.

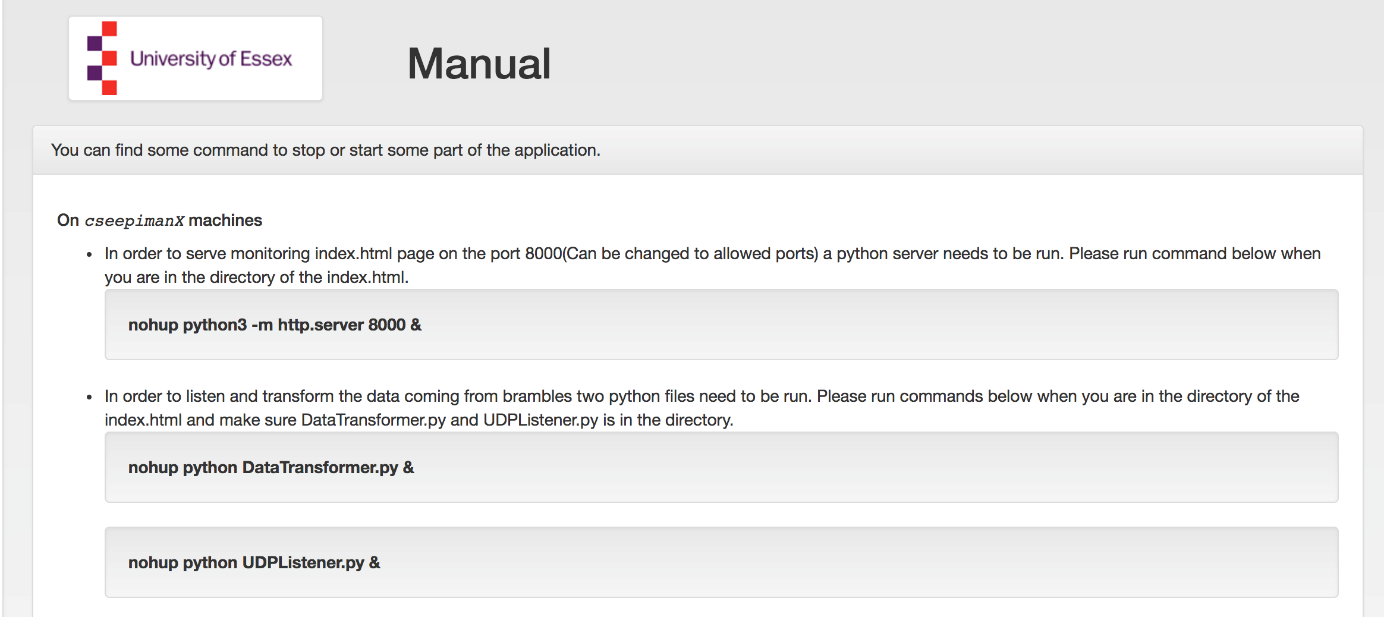


Figure - Manual Page

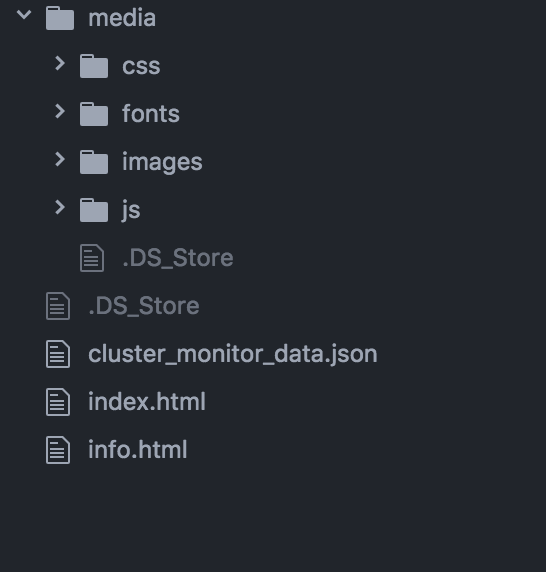
In order to keep files organised, we put every file under a tree structure. The final tree of GUI part can be seen in Figure 13.

Figure - Application Files Tree

#### Tools and Equipment

The main IDE we used to develop GUI was Intellij IDEA 2016 where many web related technologies are being supported. Intellij Idea provides also VCS integration but we found it better to use a separate program to manage version control related tasks. We used SourceTree which is free and sometimes “git” commands were enough to transfer the changes.. We also needed to generate some data for testing where we used both console python and PyCharm 2016 to generate mock data. We constantly pushed our code to GitHub with a free account plan.

# 4. Testing

## 4.1. UDP Sender

## 4.1. UDP Listener

Sprint 1

Testing the implementation of FR 3.1/ UDPlistener.py program

A GUI based software ‘Packet Sender’ (Ref : <https://packetsender.com/>) was installed and used on a remote test PC to generate test data.

It allowed simulating the process of sending UDP messages independently from the implementation of FR1 and FR2.

Results : **SUCCESSFUL**

|  |  |  |
| --- | --- | --- |
| **Test Type** | **Result** | **Details** |
| Functional Testing | **PASS** | No packet loss nor delivery failure was noted. All UDP messages correctly written to the output file. |
| Integration Testing | **not required for this sprint** |  |
| Regression Testing | **not required for this sprint** |  |
| Performance and Capacity Testing | **PASS** | No performance or capacity issues identified. |
| Compatibility Testing | **not required for this sprint** |  |
| Acceptance Testing | **not required for this sprint** |  |

**Issues raised:**

The current implementation of storing the UDP messages into a file may complicate the implementation of FR 3.3.

**Actions taken /to be taken:**

New task has been raised to investigate the potential issue and added to the backlog. To be implemented in Sprint 2

Sprint 2

: Testing FR3.3 part of DataTransformer.py program.

Testing change part of UDPlistener.py program

Results : **SUCCESSFUL**

|  |  |  |
| --- | --- | --- |
| **Test Type** | **Result** | **Details** |
| Functional Testing | **PASS** | UDPlistener.py correctly stores messages into Python dictionary.  DataTransformer.py program merges and consolidates old and new data correctly. |
| Integration Testing | **PASS** | Tested the integration between UDPlisener.py and DataTransformer.py.  DataTransformer.py program successfully loaded data from the output file created by UDPlistener.py.  DataTransformer.py program successfully loaded data from the output file created by UDPlistener.py |
| Regression Testing | **PASS** | Unchanged coding part of UDPlistenr.py still function correctly. |
| Performance and Capacity Testing | **PASS** | No performance nor capacity issues identified. |
| Compatibility Testing | **PASS** | The program runs and work the same on Unversity Lab PCs (Ubuntu) as well as CSPIMAN2 server (different version of Linux) |
| Acceptance Testing | **PASS** | Customer verified the result. |

**Issues raised:**

Panda and Jsonify libraries were not preinstalled therefore it had installed on CSPIMAN2 server. But it is only at user level which means other users will need to install them.

**Actions taken /to be taken**

Change DataTransformer.py so that the program checks whether the libraries are installed in Sprint 3.

Dr. Clark will request the system admins to install the libraries at system level.

Sprint 3

: Testing FR3.2 part of DataTransformer.py program.

Integration tests with CPU and

Results : **SUCCESSFUL**

|  |  |  |
| --- | --- | --- |
| **Test Type** | **Result** | **Details** |
| Functional Testing | **PASS** | DataTransformer.py produces a Json file in the correct format. The process loops per certain interval and produces new Json file. |
| Integration Testing | **PASS** | Integration between ‘CPUMemory\_UDPSender.c’ program and UDPlistener.py was successfully tested.  Integration between DataTransformer.py and Web application (FR4) was tested successfully. |
| Regression Testing | **PASS** | Unchanged coding parts still functioned correctly. |
| Performance and Capacity Testing | **PASS** | No performance nor capacity issues identified. |
| Compatibility Testing | **PASS** | The program runs and work the same on Unversity Lab PCs (Ubuntu) as well as CSPIMAN2 server (different version of Linux) |
| Acceptance Testing | **not required for this sprint** | Customer verified the result. |

## 4.2. Web Application

FR 1 Development

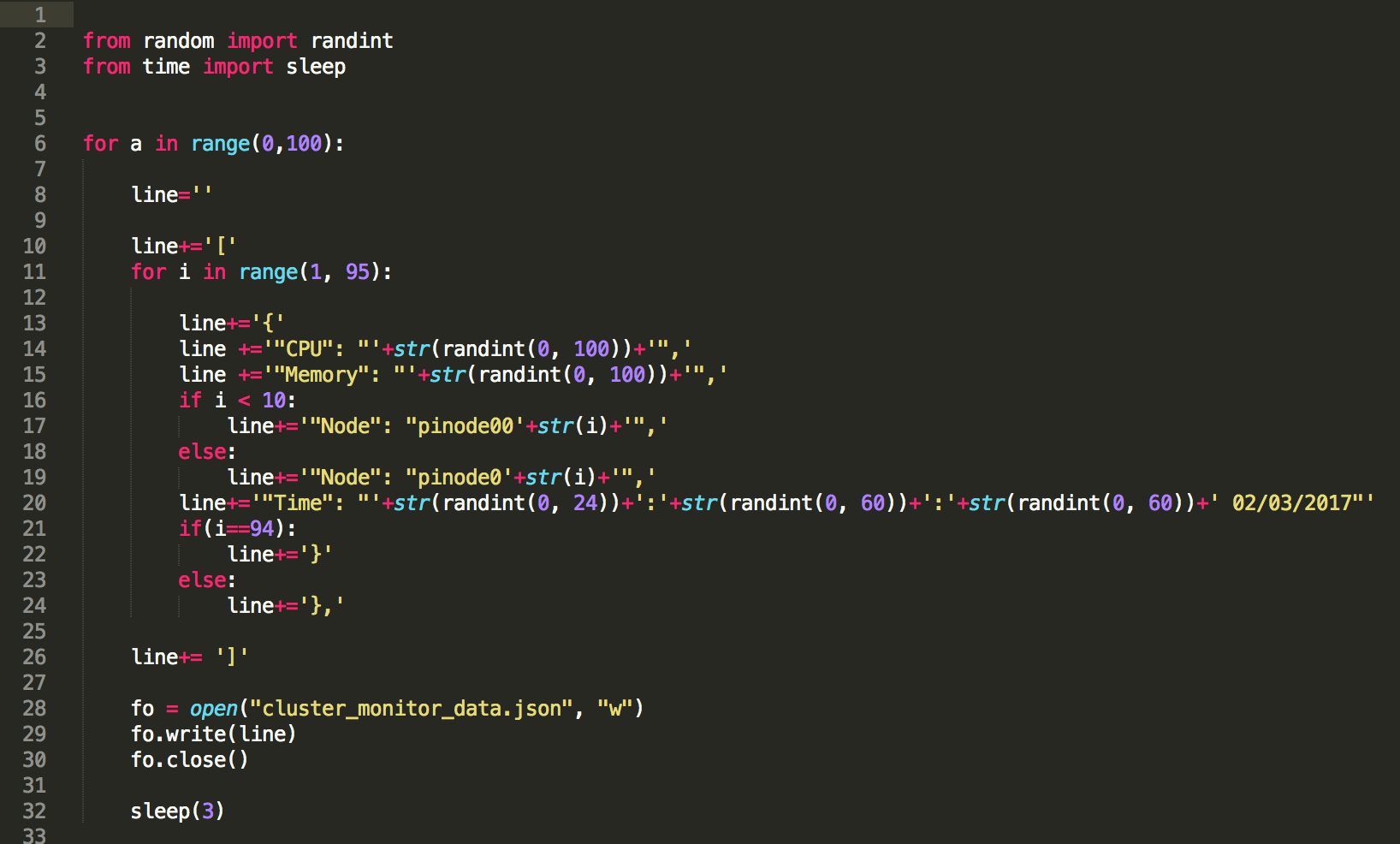
* Functional Testing: FR1 is mainly GUI part and some background daemon services that creates JSON files regularly. We constantly checked whether developed parts are working functionally as requested or not. For functional testing we created some mock data creator to see if everything works seamlessly. Mock data creator was generating the JSON file the way normally complete stack would do.
* Integration Testing: At the end of Sprint 1, we arranged a meeting to combine entire system to test whether integrated system works well or not. We did some tuning depending of customer needs e.g. refresh time of the system info.in Sprint 2 and Sprint 3. Furthermore, we run CPU testing by running some CPU consuming tasks on the Raspberry Pi nodes to see if system works as indicated in the SRS.

Figure - Mock data generator

* Compatibility Testing: During development in Sprint 1, we make sure programs needed to run Python server exist on CSEEPIMAN machines. Additionally, we needed to make sure CSEEPIMAN machines can serve a directory to a public port, in our case 8000. No further changes have been done on compatibility during Sprint 2 and Sprint 3.
* Acceptance Testing: After a completion of GUI part in Sprint 1, we checked the development and compared with SRS document. After that, we showed final product to customer for agreement. However, we need to tune refresh time during Sprint 3. No acceptance testing done during Sprint 2.

# 5. Conclusions

## Methodology/Language/Tools

Software development method employed was Agile. This approach was decided because of our regular meetings, both with the customer and as a team, as well as incremental development phases during each sprint. We used Trello to track our tasks during development. We also used a shared google drive folder to manage documentation and, as indicated in the implementation part of this document, we created a GitHub group to constantly push our code.

GitHub account can be seen here: <https://github.com/CE903GroupProjectGroup1/>

Sprint-1

The aim of sprint 1 was to build a working version of the Lightweight Monitor. We managed to finish the tasks and implement a working version in the first sprint, however some integration tasks carried over to Sprint 2 as well. We were planning to use a complete server solution for serving the index page however it turns out that a simple python http server can be enough for our purpose.

Sprint -2

We aimed to do fine-tuning and incorporate any additional requests from customer during the second sprint. We dropped the number of colours we used in the first sprint to represent freshness of the data in the cell to three colours.

Sprint -3

Last sprint was also spent for some additional tuning for entire system.

🡪 SOME info about other Parts!!!! Dunsin, Matt, Kyung

## Project Management

We researched suitable tools that would allow us to best allocate and plan tasks and workload. We allocated tasks for each sprint based on each team member’s skills. We also met regularly with the customer to ensure we were developing the product correctly.

For Sprint 1-2-3 the sprint board can be seen in the following screenshots.

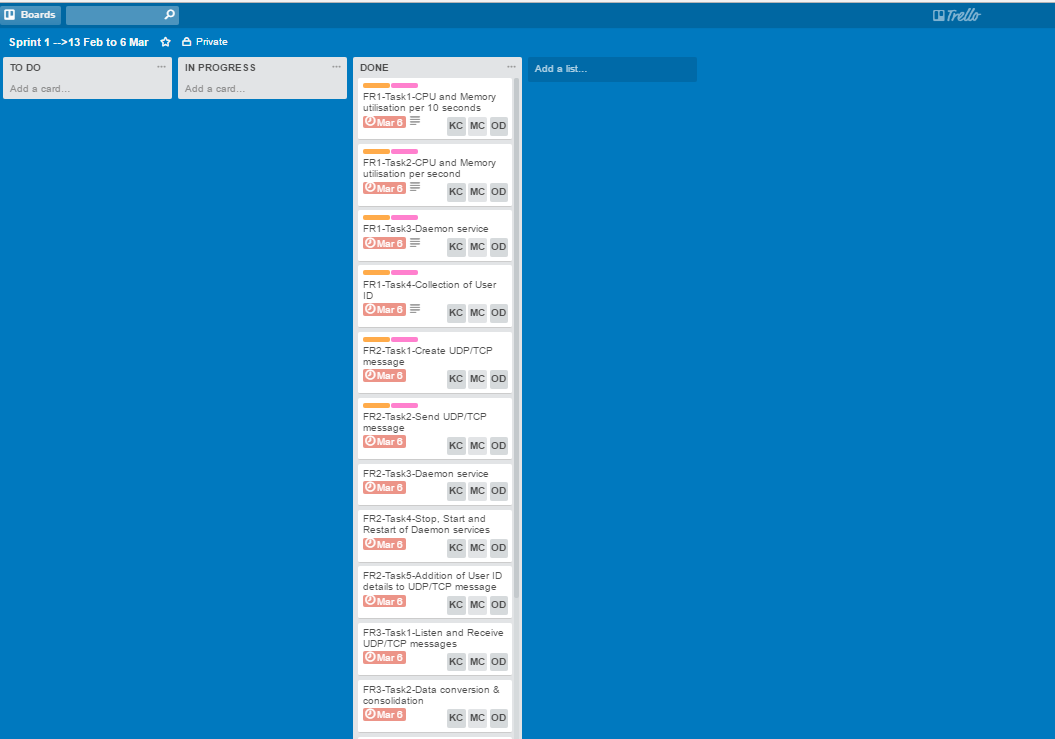


Figure - Sprint 1 Board and Tasks

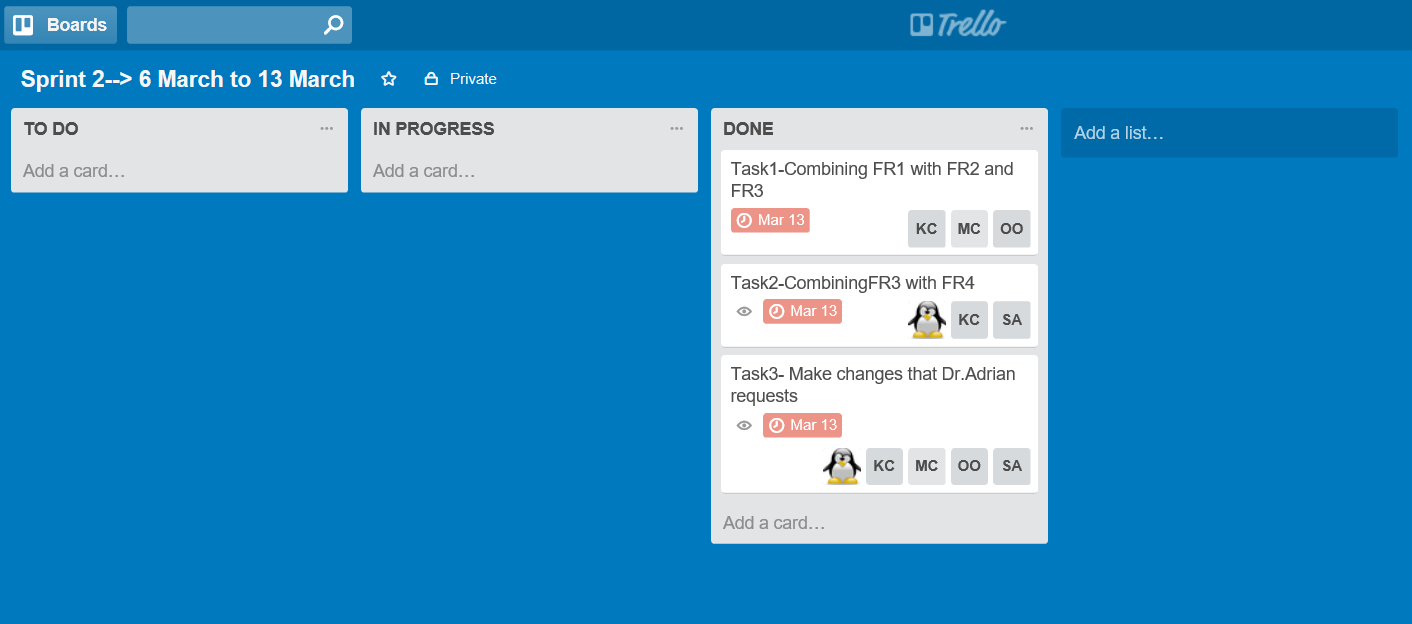


Figure - Sprint 2 board with completed tasks

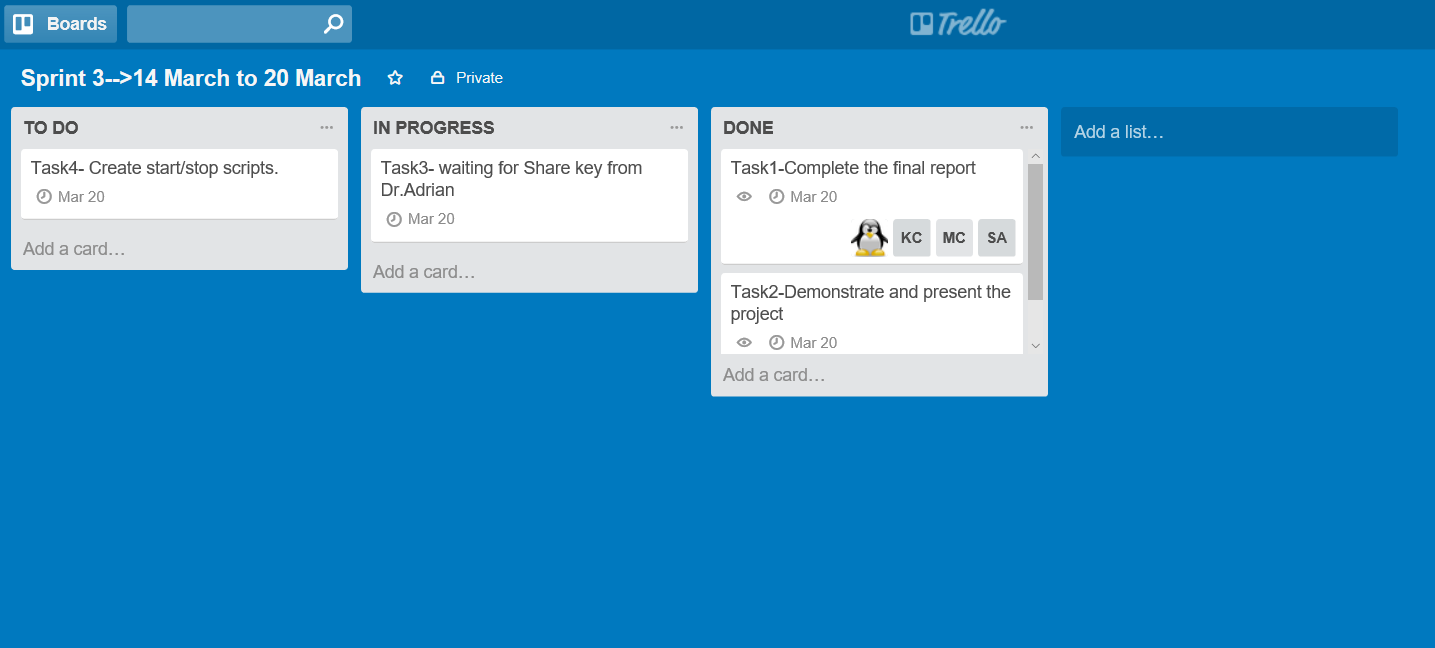


Figure - Sprint 3 board with completed and uncompleted tasks

# 6. References

[1] web reference-> <https://v4-alpha.getbootstrap.com/layout/grid/>

[2] web address reference -\_> http://www.mattkruse.com/javascript/date/