Automated Safety and Security System for the Home

Kevin Patrick Murphy

April 1, 2015

Abstract

The investigation, design and implementation of a distributed system for monitoring and controlling safety-related sensors in the home. By leveraging the areas of embedded computing, mobile applications development and server side development, the creation of a full software ecosystem was undertaken. Close attention was devoted to the software and hardware architecture required to provide optimal supporting infrastructure for the system. Configuration, communication, and alerting are the core problems addressed.

Four subsystems exist within the project, a Sensor Management System, a Peripheral Sensing System, a RESTful server side API, and an Android mobile application. At the core of each subsystem is a middleware layer providing JSON message passing services, linking all entities together.

Contents

Acknowledgments

1 Introduction

The ceiling in my kitchen is 7 feet tall, and with little ventilation, steam and smoke accumulate rapidly. This results in the smoke alarm being triggering numerous times each evening. To overcome this problem, while cooking, the batteries in the alarm are removed, and replaced when finished. However, in the event of someone forgetting to replace the batteries, there is virtually no means of alerting if a fire does occur. This began the initial conceptualization for this project.

The simplistic functionality of the smoke alarm left much to be desired. I started to devise an embedded sensor system solution that would allow complete control over alerting functionality. This lead to further ideas on how to control and manage the integrated sensors, or more specifically, giving the user the ability to configure them as required.

The main question I then asked myself was: how can safety and security be improve within the home, through the leveraging of modern hardware and software components? I kept this question in mind throughout the entire project lifecycle.

1.1 Background

Previous to starting the project I have no experience working with embedded systems, or electronics in general. Creating the circuit at the beginning of the project was one of the most challenging aspects I had to face. Without a reliable hardware implementation, the system is rendered completely futile.

Python is fastly growing as a popular multi-purpose language. It is used right across the IT spectrum, from machine learning to web development. I had never written a single line of Python before the project started, and wanted to acquire a working knowledge of the language before leaving university.

Similarly, I had only written minimal C programs previous to the project. As C++ is used for high performance programming in many areas, I wanted to gain some experience in developing with it. While the amount of C++ code in the project is small, I got to learn the basic principles of C++ development.

I had experimented with the Raspberry Pi an number of times over the last two years, but not for software development. I had previously employed it as a home media center solution. As for the Intel Galileo, I had never come into contact with one. I also had no experience with the Arduino board, so development with the Galileo started at a slow pace.

My work placement was spent at a mobile applications development company, where I worked as an Android developer. I was comfortable with the basic APIs available, but as a new version of the Android operating system was released around November of 2014, I wanted to gain experience with its new features and UI styles.

2 Problem Analysis

2.1 Internet of Things

The Internet of Things(IoT) is the term used to describe the ever increasing number of networked devices and appliances in the world around us. Employing embedded systems with networking capabilities has opened a range of opportunities, both within and outside of the home. These devices can be remotely monitored, controlled, and configured, offering ultimate freedom of how

they are utilised.

Newfound levels of automation can be achieved through the creation of intelligent systems which can learn from the vast amounts of data generated by these devices. This generated data can be used in a multitude of ways, such as to provide insightful feedback, or improve operation. The application of these systems can be seen in problem areas stretching from environmental monitoring and energy management, to air and road traffic control.

2.2 Current Safety Applications

Safety applications are becoming more prevalent in the home and in industry. Irish building regulations now require that all new houses be fitted with mains powered smoke detectors. By monitoring environmental data, early warning signs of threats can be detected, appropriate alerts made and lives can be saved.

Many companies offer monitor systems for break-ins on a subscription basis. The alerting model usually involves the detection of a breach in security, which triggers an alert to a remote operations centre, which then alerts the emergency services, neighbours or any other listed third parties. PhoneWatch offer by Eircom is an example of this type of service.

Nest Labs have created a number of embedded solutions for home automation and safety. For safety specifically, the Nest Protect system incorporates smoke, carbon monoxide, heat, and activity detection sensors. Surveillance cameras can be added also, to provide additional security. The system they have implemented is limited in terms of sensor configuration, not providing the user with any access to sensor managemnet. Their UI focuses on environmental events, such as spikes in carbon monoxide, but does not allow the checking of the actual sensor value experienced. My solution aims to display detailed sensor readings and aggregated values, which will give acute insight into environmental conditions across varying periods of time.

2.3 Achieving Safety and Security within the Home

Choosing periphery sensors to improve safety and security involved identifying the main threats which can occur within a domestic environment. I settled on the following as a basis for what my system was to mitigate against.

- Fire and Smoke
- Carbon Monoxide
- Gas leaks
- Intruders

Providing the user with appropriate feedback on current sensor levels is an essential aspect of ensuring safety. The methods of delivering this varying type of feedback depends on both the proximity of the user, and the severity of the particular sensor reading. Giving the user the ability to configure the way in which the system alerts them ensures that they consciously choose the most appropriate method of doing so. The possible methods of alerting that were considered for the project are.

- Smart phone push notifications
- An on board buzzer
- Email
- Direct Tweet via Twitter
- SMS

2.4 Embedded Computing

2.4.1 Raspberry Pi

The Raspberry Pi is a single board microcomputer, roughly the size of a credit card. It has a 700MHz Single Core 32-bit RISC ARM Processor(ARM11), along with 512 MBs of SDRAM. There are 26 general purpose input and output pins present on the board. A Secure Digital(SD) provides all storage, including the Raspbian operating system. Raspbian is a lightweight version of Debian Linux, which has been customised specifically by the Raspberry Pi Foundation. There is a full GPU on board, and a HDMI port for connection to a monitor. The Raspberry Pi model 1 B was used for the project.

The community behind the Raspberry Pi is very extensive, and having over 5 million sold since its initial release, a lot of experimentation has been done with the board. The original purpose was to provide a cost effective educational computer, but the applications of the Pi have far exceeded this.

The Pi can be up and running within 20 minutes of first receiving it. The simplistic documentation and well maintained repositories present on the Raspbian OS mean that development can begin rapidly.

2.4.2 Intel Galileo

The Intel Galileo is a single board microcomputer, slightly larger than the size of a credit card. It has a 400MHz Quark SoC X1000 32-bit single core pentium-class x86 CISC processor, along with 256 MBs DRAM and 512KB on-chip SRAM. There is no GPU present on the board. The Galileo is Arduino-Certified, and can use a version of the Arduino IDE for development. For the Galileo's operating system, a lightweight embedded Linux distribution called Yocto is used.

The community behind the Intel Galileo is considerably smaller than that of the Raspberry Pi. When dealing with problems with the Galileo board, it is much more difficult to find online solutions or tutorials. Intel maintain two repositories for the Yocto OS, one with preinstalled development software, and the other without. Getting setup with the Galileo takes some additional time as there are many options for connecting to it: direct ethernet, serial, or USB.

2.5 Functional Specification

An embedded sensor system and Android smart phone application which supplies the user with live environmental data and security services.

The following sensor modules will be included:

- MQ-7 Carbon Monoxide Sensor
- MQ-2 Flammable Gas & Smoke Sensor
- Infrared Motion Detection Sensor
- Thermistor for Temperature levels
- Buzzer for immediate vicinity alerting
- Raspberry Pi HD Camera module for Image capture and live video streaming

This application will take advantage of the latest Android APIs, and follow modern UI and UX guidelines.

2.5.1 Functional Requirements

The system should be configurable by the user, allowing the control of each individual sensor, data management, alerting services, and general system details.

The system should have the ability to alert the user remotely.

The gathered sensor statistical data will be graphed for the user to monitor readings over extended periods of time.

Context sensitive feedback will be provided to the user on current sensor readings.

The system should include a mechanism for direct communication, allowing the system to operate in environments where network connectivity is limited. All original functionality should be available when directly communicating.

The system should provide a device authorisation process, where one single user can have full administrative control, and allow all others only limited access.

The system should maintain a database table for general system details.

The system should have a interface for connection with additional sensing devices.

2.5.2 Non-Functional Requirements

The system should be operable irrespective of network connectivity.

The implementation should be extendable and allow for easy inclusion and integration of any desired sensor module.

Consideration should be give to optimisation, in order to reduce CPU load and maximise speed.

Interaction with sensor modules in particular will require close attention to ensure efficient and accurate readings are obtainable.

The system will be powered via the mains, but attention should be given to maximizing efficiency to allow the move to alternative power sources in the future, i.e. solar or wind.

Care should be taken when designing the data layer of the system, ensuring optimal memory management.

3 Design

3.1 Development Approach and Methodology

In a development project of this scale and time frame, a hybrid development approach of Agile and Rapid Application Development(RAD) was determined to the most suitable. This hybrid model offered flexibility for the system to change and evolve during its design and implementation, while also providing working software early on. During the implementation phase of development the prototyping aspects of RAD were utilised for the feasibility testing of design decisions.

From the beginning of the project it was evident that a process heavy methodology to development would constrict the time frame of the systems overall development. Applying an iterative and incremental agile approach reduced the time spend in planning, allowing a longer development phase. This is not to say that planning played a lesser role in the systems life cycle, but that essential core functionality was defined and a rudimentary version of the system designed early in the project's life cycle. Using this initial core design, implementation of the subsystems began, and in parallel with their development, any desired additional functionality or feature was incrementally designed and integrated.

3.2 Architecture

3.2.1 Hardware

The final hardware architecture of the system comprised of a Raspberry Pi, with four sensors attached; a thermistor for temperature readings, an MQ7 carbon monoxide sensor, an MQ2 flammable gas sensor, and a passive infrared motion detection sensor. Also connect to the Raspberry Pi was a high definition camera module, a wifi dongle and a buzzer for alerting the user.

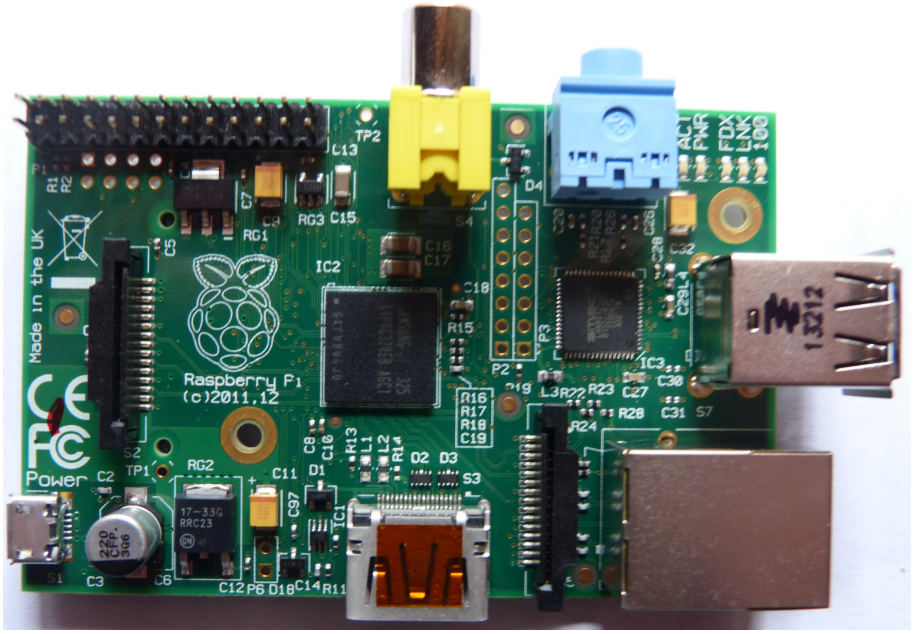

Figure 1: Raspberry Pi board

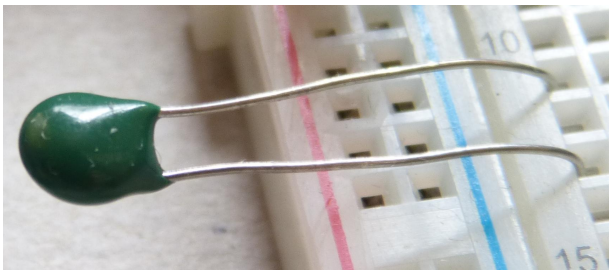

Figure 2: Thermistor sensor module

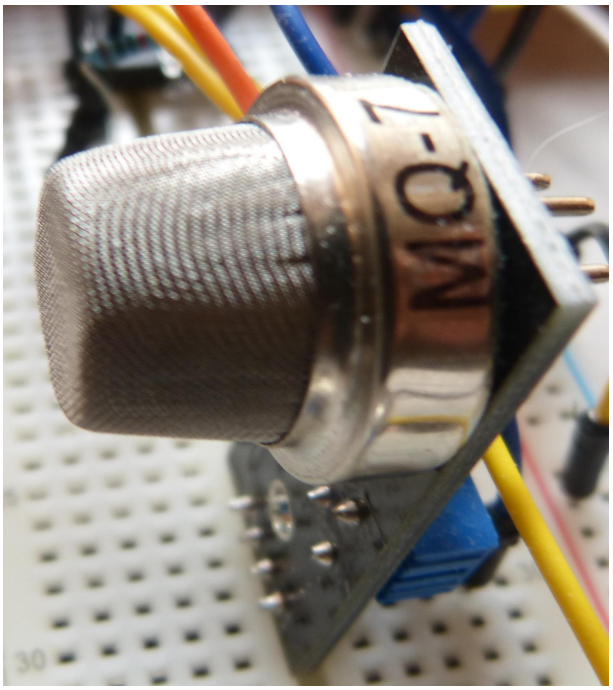

Figure 3: MQ7 carbon monoxide sensor module

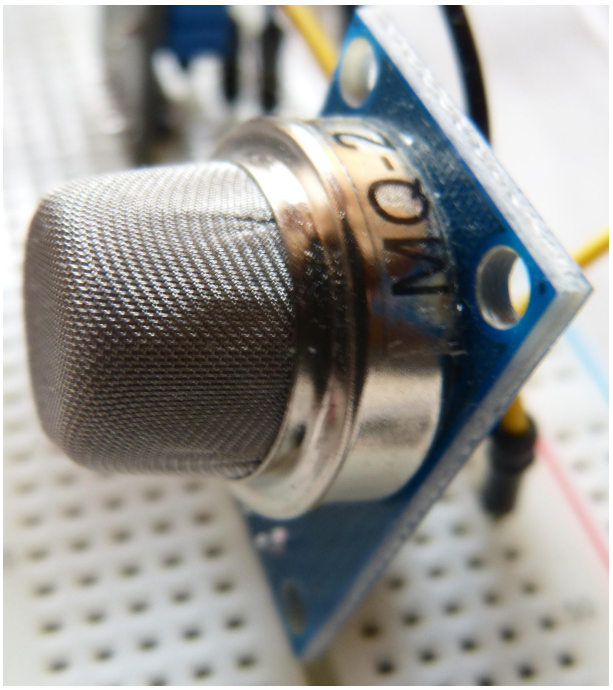

Figure 4: MQ2 flammable gas sensor module

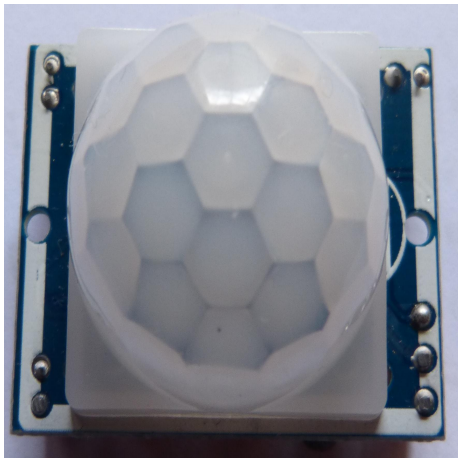

Figure 5: Passive infrared motion detection sensor

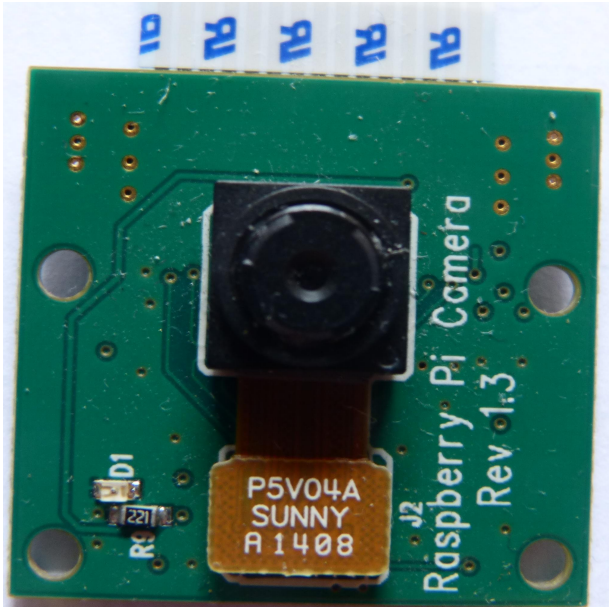

Figure 6: HD camera

Figure 7: Broadcom Wifi dongle

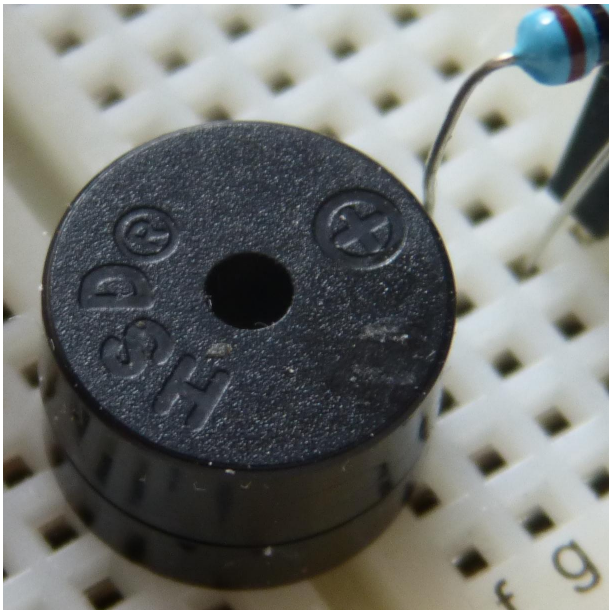

Figure 8: Buzzer

3.2.2 Software

The final software architecture was composed of three main components, with the addition of a peripheral sensing node.

- The Sensor Management System physically running on the Raspberry Pi
- The server side RESTful API running on CS1
- The Android application
- Peripheral sensing node running on an Intel Galileo

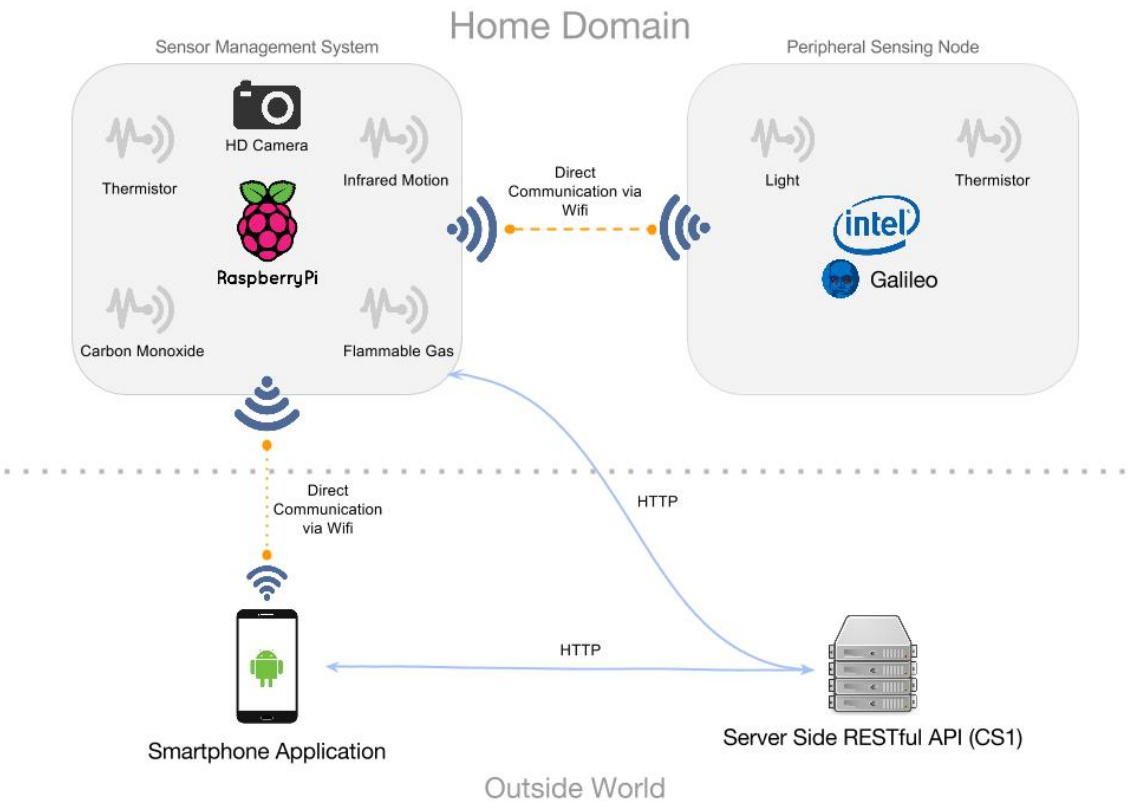

Figure 9: High level system architecture diagram

Sensor Management System
This will form the core of the entire system, managing all sensor module interaction. Sensor value monitoring and alerting service triggering should take place natively on the system.

The system will allow direct connection via the Android application. To accomplish this, the system should maintain its own private wireless network, in the form of a non-internet facing access point(AP). A protocol for establishing connection and full-duplex communication from the system to the Android application will be designed, allowing sensor data transfer and system configuration.

Three local database tables will be required for operation:

- Sensor values table
- System meta data, including name, location, approximate and GPS coordinates
- System admin details for authentication and control

RESTful API
This will form the server side backend of the system, and provide the application programming interface(API) for both the Sensor Management System, and the Android application. Functionality includeds persistent cloud storage and retrieval of sensor readings, camera assets management, and remote configuration of the Sensor Management System.

Android Application
This will form the user interface of the overall system. The application will provide sensor output

levels, a gallery of camera stills and videos, graphed statistical data on previous sensor readings, and a configuration interface for the system.

Related safety feedback information should be delivered to the user via the app. A dataset of safety metrics linked to the sensor output readings will be developed offering the user real time advice in plain english.

The app will utilise the server side RESTful API as its main communication intermediary to the system, but will also have the ability to connect directly, allowing it to read the sensor data straight from the system, and update the systems configuration.

Push notification functionality should also be include to provide the user with remote alerting.

3.3 Design Decisions

The initial software architecture was for a self contained system with each component running natively on the Raspberry Pi. While it is a very capable device, as more computational power was required for the operation of the system, the feasibility of running a server in parallel came into question. Using the Glances Python utility [1] I was able to actively monitor the CPU performance of the Raspberry Pi while the system was in operation, which consitantly remained above 60%, even in the early stages of development. In order to safeguard against congestion and drops in overall performance, an alternative solution was devised to reduce load on the Raspberry Pi. This lead to the adoption of a client-server architecture involving the system periodically sending data to CS1, where it could be stored, and delivered to smart phone applications, greatly reducing the computational load on the physical device.

A Representational State Transfer(REST) architecture was chosen to provide the server side back-end functionality required to service both the Sensor Management System and the Android smart-phone application. REST is a lightweight alternative to a Simple Object Access Protocol(SOAP), which utilised the Web Services Description Language(WSDL), an XML based interface definition language. The advantages of REST are its extensibility, simplistic design and portability.

The user Interface for the system was initially in two parts, a web interface for graphing the collected sensor data, and the Android application, for current readings, control and alerting. As the project progressed, the choice was made to unify the UI, providing all functionality in the Android application.

In order to maximise performance, every subsection of the system was multithreaded where possible. With this decision came the added complexity of appropriate thread management and synchronization. The Raspberry Pi has a single core, so no substantial benefits were apparent when developing the system, but in the eventuality of migrating to more powerful hardware will result in instant performance improvements.

In order to maximise sensor performance, all interaction directly with sensor modules is implemented in C, with a C++ infrastructure to avail of basic object oriented principles such as inheritance and extensibility. This was chosen to reduce the CPU load when probing sensor modules. In order to utilize the C++ implementation, it was necessary to develop a Python wrapper, providing the ability to access sensor values at greater speeds than a standalone Python implementation.

A core feature of the overall system is providing the user with rapid alerting. With that in mind, it was decided that alerting should be handled natively in the sensor management system. The alternative was to perform safety checking server side, but this may lead to alert latency.

[1] http://pypi.python.org/pypi/Glances/

3.4 Data Storage

MySQL was chosen as the Relational Database Management System(RDBMS). While its use and interaction is facilitated in most modern programming languages, the real benefit comes from its aggregation functions. Having the ability to select maximum, minimum, and average values from the database in a concise SQL statement greatly reduces the amount of code required, and as rows can be indexed and prepared statements optimized, overall performance is booted also.

Using a NoSQL database was considered as the performance and scalability benefits can be substantial when compared with generic SQL databases. NoSQL databases required a substantial amount of code to be written for interaction and aggregation, something which comes as standard in all SQL storage systems. This deterred adoption.

The Raspberry Pi is running an instance of the MySQL RDBMS, while the RESTful server side system is maintaining a database of tables which is identical to those present in the natively running instance. This provides data redundancy, and in the eventuality of the server side system becoming unreachable no data will be lost.

3.5 Data Modelling

As memory is a constant constraint when dealing with an embedded system, consideration was put into optimising the structures of the tables within the RDBMS. Due to a lack of time when implementing the system, only one table was eventually used for sensor readings, however the original design contained three tables to minimise memory usage. A table for all current day sensor readings, another table for aggregated readings per hour of each day, and a final table for a summary of aggregated readings per day. The data was then to be aggregated automatically at a given time interval, and reduced into the subsequent table. This way, over time the data growth would be limited.

A table was created to storage the systems details, such as its user specified name, location, and GPS co-ordinated.

For push notifications, it is necessary to maintain a table for all registered device identification numbers. These IDs are to be stored via the RESTful API and are used when sending out a push notification to a device.

3.6 Communication Design

Developing a distributed architecture involved designing a message passing mechanism utilising Javascript Object Notation(JSON), a lightweight data-interchange format. Its native support in most modern programming languages and easy to manipulate syntax facilitated fast integration in each subsystem.

XML was considered, but its use is more applicable to heavily structured documents. Library support for object serialization to JSON greatly promoted rapid application development, providing the ability to easily communicate current sensor readings and system configurations across a network.

A standard structure was defined for the JSON messages being communicated.
The basic structure was as followed:

Listing 1: JSON Message Structure

```
{
    service    : "Requested Service",
    payload    : "Data to be transmitted"
}
```

The service key indicating the service being requested, and the payload containing the data to be acted upon.

3.7 User Interface

The Android smart phone application is the only user Interface(UI) for the project. The platform provides a UI framework and a large amount of extensible UI view components, which can be combined with ease. The Android OS offers native animations and effects for improving user experience(UX), allowing the creation of visually responsive features with little effort.

The UI was designed with simplicity in mind. The clear and concise presentation of data to the user was the main priority. To achieve this, core aspects of functionality were designated individual screens within the app.

The UI is composed of 4 screens. The initial screen where the latest sensor data is presented. The camera screen, where captured images and videos are presented. The graphing section, were line charts display sensor value statistical information. And the final section is the control and configuration options, where the system can be reconfigured.

As smart phone application has the build in functionality for communicating directly with the sensor management system, it was necessary to design the UI to respond accordingly to how the application is receiving its data.

3.8 Alerting

The method of remote alerting chosen was push notifications, via the Google Cloud Messaging(GCM) Service.

The GCM is a free service that enables developers to send downstream messages (from servers to GCM-enabled client apps), and upstream messages (from the GCM-enabled client apps to servers). This could be a lightweight message telling the client app that there is new data to be fetched from the server (for instance, a "new email" notification informing the app that it is out of sync with the back end), or it could be a message containing up to 4kb of payload data (so apps like instant messaging can consume the message directly). The GCM service handles all aspects of queueing of messages and delivery to and from the target client app.[2]

4 Implementation

4.1 Tools and Techniques

Each component of the system was implemented using a different programming languages, with the exception of the Principal Sensing Node. As a result, each was developed using a different set of tools, techniques and libraries. The reliance on Integrated Development Environments(IDE) to

[2]Description taken from http://developer.android.com/google/gcm/gcm.html

increase productivity and mitigate against syntactical errors was paramount to the rapid development of the project.

4.1.1 Sensor Management System

Implemented in Python, a very flexible and easy to use dynamically typed language. Its succinct nature and dynamic typing aid rapid application development. The community support behind the language also greatly encouraged its adoption for this section of the system.

To implement the sensor system, I used the PyCharm IDE. This provided real time code validation, automated code generation and syntactical checking.

4.1.2 RESTful API

Implemented in the server side language PHP. PHP offers a lot of server side functionality as standard, and required no third-party libraries to facilitate the features that were required. This will prove beneficial in the situation of migrating servers.

Deployed on the universitys Apache web server, CS1. Utilizing this pre-existing infrastructure resulted in less overhead when initially starting the project, allowing the development phase to begin earlier.

4.1.3 Android Application

Native Android applications are implemented in Java, and use XML for specifying their user interfaces. It is possible to design applications in HTML and CSS, embedding them in the application within a WebView, which handles rendering, but this greatly hinders performance, and limits the use of the native device APIs, such as location services and animations.

Android Studio, the officially supported IDE for Android development, was used to develop the application. It is also based upon the Intellij platform of IDEs, and provides a graphical interface for developing user interfaces.

4.1.4 Peripheral Sensing Node

Python was also used when developing this additional entity of the system. This made interoperability more straightforward when integrating the two sensing systems together. Benefits also included code reuse for the previously written for the Sensor Management System.

4.2 Sensor Management System

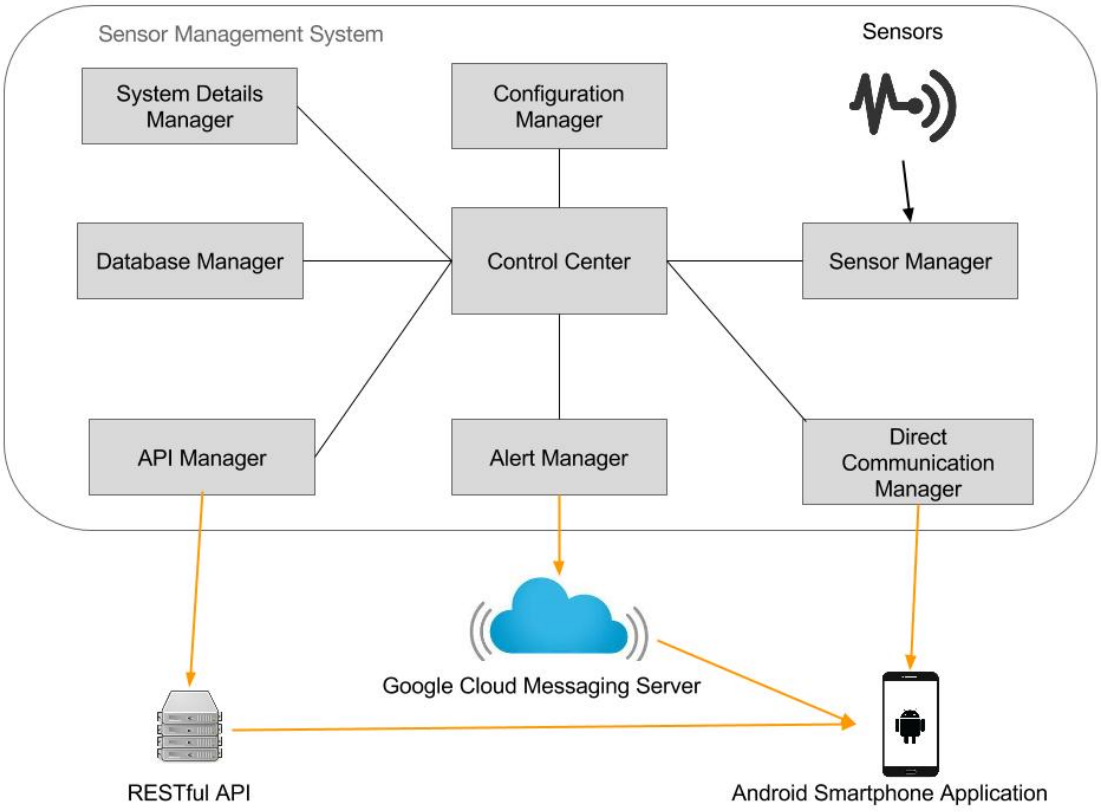

Figure 10: Sensor management system high level class diagram

4.2.1 Sensors and Peripheries

To control sensors connected to the Raspberry Pi, the open source WiringPi C/C++ general purpose input/output(GPIO) library was employed. It provides basic interaction mechanisms for both analog and digital sensors. The community support for this library is substantial, and has become the de facto standard library for embedded Raspberry Pi projects.

A limitation of the Raspberry Pi is that is does not provide any analog GPIO pins. To overcome this, an additional analog to digital converter(ADC) was added to the systems circuitry. The MCP3008 ADC, an 8 channel 10-bit analog input, was chosen due to its support in the WiringPi GPIO code library. This facilitated easy access to any analog sensor that needed to be included.

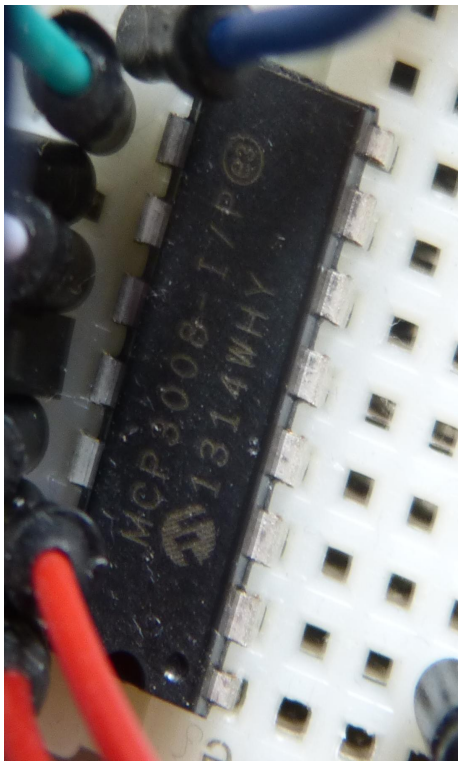

Figure 11: MCP3008 analog to digital converter

Each individual sensor required a slightly different code implementation.

The MQ7 carbon monoxide(CO) analog sensor needs a heating cycle of 5v for 60 seconds, and 1.3v for 90 seconds continuously for 48 hours, until a true value of 0 is output. This is due to the sensor being sensitive to other chemical elements, in particular hydrogen. The heating cycle is designed to burn off excess molecules touching the senors filament, resulting in more accurate readings.
While in the heating cycle, the sensor is capable of detecting variances in CO levels. In order to provide the user with accurate data, an algorithm was defined to monitor differences in CO levels. By recording previous values, tracking current value increases or decreases, and presenting the user with the difference in values, allowed accurate CO readings from the moment the system is started.

The MQ7 flammable gas analog sensor can detect liquefied petroleum gas, i-butane, methane, alcohol, Hydrogen, and smoke. It also required a heating cycle identical to the MQ7.
The passive infrared motion detection digital sensor measures infrared (IR) light radiating from objects in its field of view. It can detect humanoids/pets from up to 20 feet away.

A passive analog thermistor was chosen to provide temperature readings.

A passive buzzer was included in the systems circuitry also, to provided local proximity alerting.

To capture visual data, the high definition Raspberry Pi camera module was included. As this module is produced by the developers of the Raspberry Pi board, very little code implementation is required to perform interaction, and two comprehensive libraries, Raspivid and Raspistill, are natively available to handle video and image capture respectively.

4.2.2 Sensor Management

Within the system, each sensor was defined within its own individual class. Abstracting individual sensors like this allows for granular configuration, and the delegation of alerting responsibility to the individual module. Each sensors has an alerting threshold, which when met will perform the necessary alerting functionality. The rate at which the sensor is probed is also configurable. Each sensor has its own priority that is used for multithreading, and also in the eventuality of the system becoming alternatively powered, it would allow for graceful degradation as power levels diminish.

The Sensor Manager class see in Figure 10 facilitates high level sensor orchestration. Its in-charge of data collection, and thread management. A thread is created for each sensor, which is scheduled to run in the interval denoted by its probe rate. It is not necessary for all sensors to be updated constantly, by implementing the probing in this way, safety critical sensors, such as the MQ2 Flammable Gas sensor, can be configured to update its value every second, while a non-safety critical sensor, such as the thermistor, can be probed at longer intervals. In the eventuality of changing the power source of the system, it would also facilitate the marshaling of power exhaustive sensors, allowing dynamic scheduling as resources diminish.

4.2.3 Multithreading

The Raspberry Pis processor, the ARM1176JZF-S, has a single core. While it is not a true multithreaded environment, the decision to multithread the system was taken to ensure a high level of portability. Over the course of this project a new Raspberry Pi model was released. The new model contains 4 cores, and an increase amount of RAM, of 1GB. The multithreading already featuring in the system would allow for swift migration to this new Raspberry Pi model, and instant performance benefits.

4.2.4 System Configuration

In order to make the system fully configurable, an abstract base class, called Configurable, was implemented, which all manager classes in Figure 10 inherit from. This class provides the layer of abstraction needed to dynamically update the state of each class. The system does not required halting when reconfiguring.

The Javascript Object Notation(JSON) was utilised as the means for specifying the required state configuration of individual classes. The Configuration Manager was delegated the responsibility for administering the updating of the system‘s configuration. Listing 2 shows the default JSON configuration structure. It contains an object for each manager class present in the system, and an array of sensor objects with individual state values for each.

The System is configurable form two sources, via the RESTful API, or direct from the Android Application. To achieve remote configuration, the system periodically polls the RESTful API for an updated configuration file. If present, it requests the raw JSON string, parses it, and then reconfigures the system accordingly. The time interval at which the system polls is also configurable. Via direct communication, the user can push a new configuration at their own discretion.

Listing 2: JSON Configuration Structure

```
{
  "alert_manager": {
    "buzzer_on": false,
    "camera_on": true,
    "lockdown_on": false,
    "push_on": false,
    "video_mode": false
  },
  "api_manager": {
    "camera_image_upload_rate": 60,
    "sensor_value_upload_rate": 30,
    "sys_config_request_rate": 60
  },
  "sensor_manager": {
    "collection_priority": 1,
    "collection_rate": 15
  },
  "system_details_manager": {
    "gps_lat": "Not set",
    "gps_lng": "Not set",
    "location": "Sitting Room",
    "name": "Kevin's Safety System"
  },
  "wifi_direct_manager": {
    "sensor_value_send_rate": 10
  },
  "sensors": [
    {
      "name": "carbon_monoxide",
      "alert_threshold": 50,
      "is_active": true,
      "priority": 1,
      "probe_rate": 10
    },
    {
      "name": "flammable_gas",
      "alert_threshold": 50,
      "is_active": true,
      "priority": 1,
      "probe_rate": 10
    },
    {
      "name": "temperature",
      "alert_threshold": 50,
      "is_active": true,
      "priority": 1,
      "probe_rate": 10
    },
    {
      "name": "motion",
      "alert_threshold": 1,
      "is_active": true,
      "priority": 1,
      "probe_rate": 10
    }]
}
```

4.2.5 Alerting

The system alerts in two ways. Proximity altering in the form of a physical buzzer present on on the circuitry, and direct mobile device push notification alerting via the Google Cloud Messaging(GCM) Service. The system can be configured to be in a lockdown state, where in the event of the infrared motion detector detecting activity, the on board camera is activated.

Each sensor maintains its own alerting threshold, and when reached it performs its defined behaviour. For example, when the carbon monoxide sensor‘s threshold is reached, a push notification is sent, and the physical buzzer attached to the systems circuitry is sounded.

To send a push notification, a registration process must first take place between the Android application and the GCM service. This process is described in Android application implementation section 4.5.7. Resulting from this registration is a unique device identification number, which must be persistently stored. In this implementation, the registration ids were stored by the RESTful API, and retrieved prior to sending each push notification.

The registration id is sent along with a JSON formatted payload via a HTTP request to the GCM service, which handles delivering the payload to the respective device.

Listing 3: JSON Push Notification Payload Example

```
{"sensor" : "thermistor", "value" : 100, "location": "Sitting Room"}
```

The key ”sensor” identifying the individual sensor module, the key ”value” denoting the current sensor reading, and the ”location” key is the user specified location of the system which has invoked the sending of the notification.

In the implementation, the Alert Manager is delegated all alerting responsibilities. Other than push notification requests, it also handles sounding of the buzzer, and camera operation.

4.2.6 Video and Image Capture

The Raspberry Pi has an on board Mobile Industry Processor Interface(MIPI) compliant Camera Serial Interface. The HD camera module, also developed by the Raspberry Pi Foundation, is attached directly to this CSI port. The module has a five megapixel fixed-focus camera that supports 1080p30, 720p60 and VGA90 video modes, as well as still image capture. It attaches via a 15 cm ribbon cable. To interact with the camera, two command line tools are present natively in the Raspian OS, Raspstill and Raspvid. These allow still image and video capture respectively.

The system performs the following tasks with the camera module.

- Still image and video capture, uploaded to the backend server, or delivered directly to the Android application via a Secure File Transfer Protocol(SFTP) server running on the Raspberry Pi
- Local and Remote video streaming over HTTP

To perform video streaming, the open-source VLC Media Player was chosen to provide the necessary functionality. Inbuilt within VLC is a streaming media server, which can be invoked programmatically, and allows for the specification of a wide range of media encoding, streaming, and networking options.

When called, the system simple uses the Raspvid command line utility to start video capture,

and pipes the output to the VLC media streamer, which then constructs the necessary infrastructure for the network stream.

When directly communicating, the Raspberry Pi has a Secure File Transfer Protocol(SFTP) Server running as a daemon which allows protected access to the captured images and videos. This is later utilized programmatically by the Android application.

4.2.7 API Communication

The API Manager class orchestrates all network communication with the RESTful API. The API Manager performs the following tasks:

- Uploading of current sensor values
- Requesting of updated system configurations
- Uploading of images and videos
- Retrieving of device registration ids for push notification alerting

Each of theses operations is scheduled in its own thread, making their timing individually configurable. All communication with the RESTful API is carried out via the HTTP POST method.

To identify the service requested when communication with the RESTful API an additional HTTP header called "service", is added to the request. This is parsed server side and the corresponding functionality executed. There are efficiency benefits with this approach in the cases of checking for updated configuration or requesting push notification device ids, as no data is sent in the request body, minimising the overall request size.

An example of the request body for sensor values to be uploaded to the RESTful API can be seen in Listing 4 below. As the service being requested is contained in the request header, the content body is minimal in size and complexity.

Listing 4: Sensor Values Request Body Example

```
{'motion': 1, 'flammable_gas': 0, 'carbon_monoxide': 80, 'temperature': 65}
```

With every HTTP request, there is a subsequent HTTP response. To minimise the overall number of HTTP requests needed for operation, additional information is passed back to the Sensor Management System in the response payload. Standard fields in the payload include a status code for reactive error handling, and a timestamp for monitoring latency. Additionally included is a flag for indicating which camera action is being requested. This allows for the requesting of remote video streaming, or still image capture without the need for additional API calls.

Incorporated in the class are a number of fault tolerance mechanisms to protect against unexpected network loss. By employing Pythons Try/Except functionality, this class can gracefully fail when network connectivity is unavailable, and automatically continue performing its tasks once connectivity is re-established.

4.2.8 Direct Communication

The system required the ability to directly communicate to the Android application, allowing operation in situations where there is no available connection to the internet.

Originally Wifi Direct was chosen as the means for direct communication. The Wi-Fi Direct

certification program is developed and administered by the Wi-Fi Alliance. Wifi Direct allows for communication without an access point, and can cater for multiple peer connectivity at normal Wifi speeds. It works by creating a software access point(Soft AP) within the device that supports Direct. This Soft AP provides a Wifi Protected Setup(WPS) connection.

The Broadcom Wifi Adaptor designed to integrate seamlessly with the Raspberry Pi was chosen to allow the inclusion of Wifi Direct in the system. However, due to limitations of the Adaptors firmware not supporting Wifi Direct at the time of development, an alternative solution was sought.

To accomplish this feature, an access point(AP) administered by the Raspberry Pi was implemented. This also required the Raspberry Pi to run a Dynamic Host Configuration Protocol(DHCP) server, to automate connected device IP allocation. The AP was configured to be completely local, and not forward any traffic to the wider internet.

To connect to the system, firstly the device must be connected to the Sensor Management System‘s AP. Next the device must initiate a connection handshake by sending a connection message to identify itself and specify the type of device it is. The identification is handled differently depending on the type of device. For an Android device, it uses a unique 64-bit hex string device id which is obtained directly from the Android OS. For peripheral sensing nodes, a 48 bit integer representing the device‘s MAC address is used..

The Sensor Management System maintains a multicast channel, on a pre-specified IP address and port number. It listens on this IP address for connection messages, and drops all but those which conform to the defined format, seen in Listing 5.

The message in Listing 5 is sent from an Android device to the specified multicast channel IP address, where the Sensor Managment System is listening. A connection verification message is then sent back to the Android device, and if the connection was successful, data passing begins. Alternatively, if unsuccessful, pairing must be reinitiated by the Android device.

Listing 5: JSON Connection Message Object

```
{
   "service":connect,
   "payload":{
      "session":{
         "device_id":e1cadbafc6804e3f,
         "ip_address":192.168.42.2,
         "time_stamp":2015-03-09 20:23:26,
         "type":android
      }
   }
}
```

Once the device and system are ready, all further interaction is handled via direct TCP socket communication.

An additinal benefit of implementing device connection in this way is that multiple devices can join a multicast channel, forming a group. This facilitates the communication of messages to all connected devices easily. As new devices connect, all nodes on the network listening on this multicast socket can react accordingly. In the case of alerting, if a sensing device experiences conditions that warrant an alert to be issued, a message can be pushed to this multicast channel, and all devices listening can sound their alarms.

The Direct Communication Manager class encapsulates all the above system functionality. It pe-

riodically pushes data to the connected devices, and acts as an arbitrator for all incoming messages.

The multicast IP range of 224.0.0.0 to 239.255.255.255 is specified in RFC 5771, an Internet Engineering Task Force (IETF) Best Current Practice document (BCP51) [3]. The IP address 224.1.1.1 was used in the implemetation for the multicast channel.

The obvious shortcoming of this implementation is that the device must be connected to the system‘s AP, disconnecting it from the wider internet. In terms of user experience(UX), a better solution would be to use Bluetooth Low Energy for application to system communication.

[3] http://tools.ietf.org/html/rfc5771

4.3 Peripheral Sensing Nodes

4.3.1 General Operation

A scaled down version of the Sensor Management System was designed and implemented to act as an additional Peripheral Sensing System within the home. Due to time limitations, the Peripheral Sensing System implementation is focused on gathering sensor data, and relaying it to the Sensor Management System. There is not persistent storage of sensor readings, and the system is static in terms of user configuration.

Two threads exist in the Peripheral Sensing System, both are used for communication with the Sensor Management System. One thread is purposed for gathering and sending sensor data, the other thread is for receiving data.

4.3.2 Hardware

For this project, the Intel Galileo generation 1 was used as the hardware for the peripheral node. The Galileo was chosen to highlight the Sensor Management Systems ability to manage a heterogeneous network. The Grove Starter Kit Plus - Intel IoT Edition provides the base shield and sensors, which have a modular plug-and-play design.

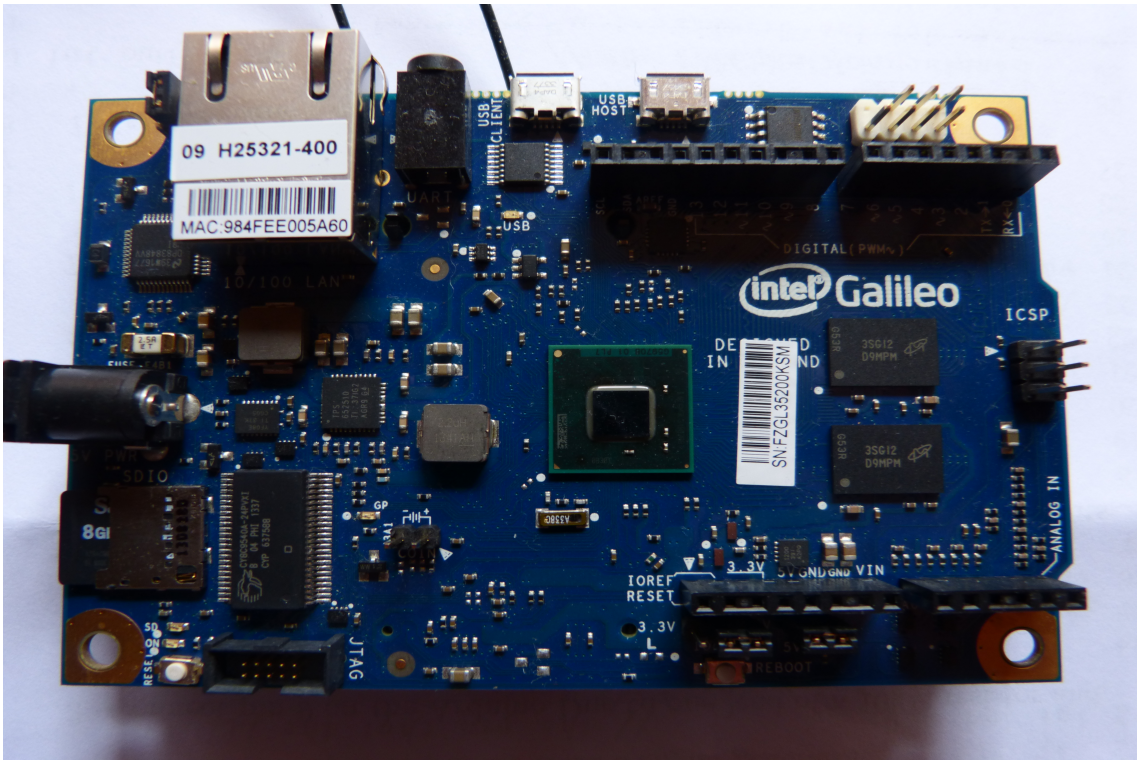

Figure 12: Intel Galileo gen 1

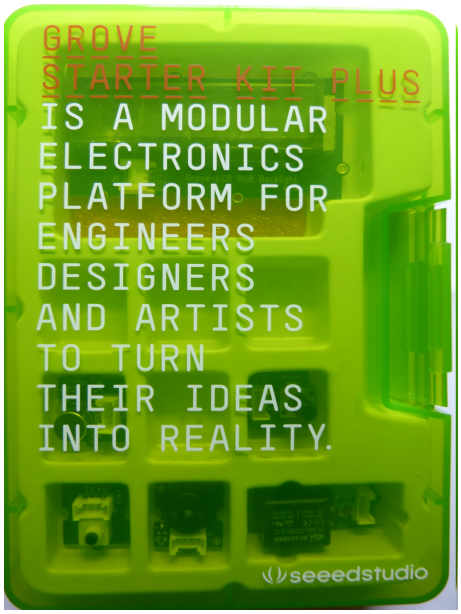

Figure 13: Grove IoT sensor kit

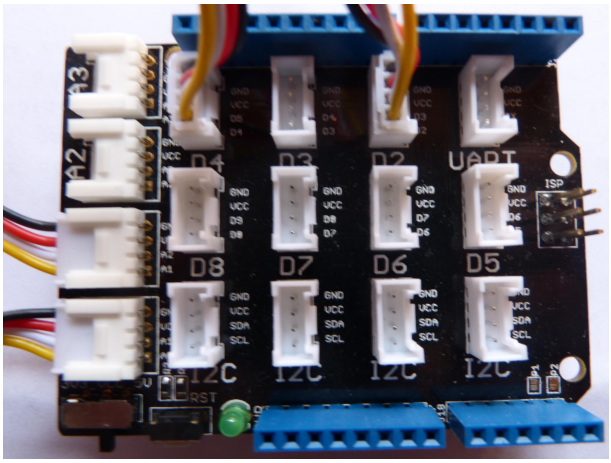

Figure 14: Grove base shield

Attached to the Galileo, via the Grove base shield, are an LED, a temperature sensor, a light sensor, and a touch sensor.

The touch sensor was included to demonstrate remote alerting, as when it receives input, it sends a message to the core Sensor Management System, which sounds a buzzer. Similarly, the LED was included to demonstrate message passing in the opposite direction, from the core system to the Galileo.

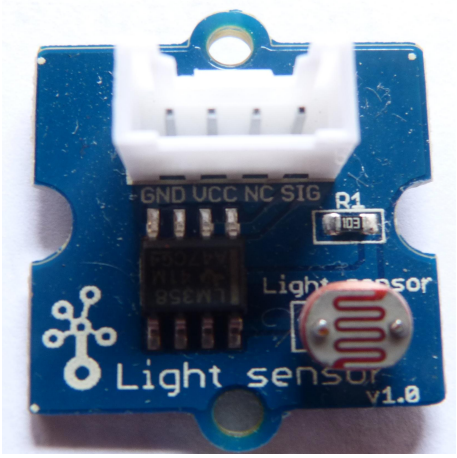

Figure 15: Grove light sensor module

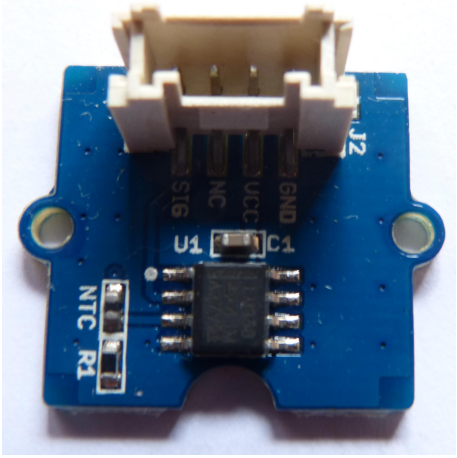

Figure 16: Grove temperature sensor module

In order to avail of high level software, such as Python and Git, it was necessary to boot the Galileo from an SD card with the Yocto system image. Yocto is a lightweight version of Linux, which is specifically purposed for embedded systems.

4.3.3 Communication with Sensor Management System

The core Sensor Management System provides the communicational infrastructure to allow any device capable of connecting to the system‘s AP to become a peripheral sensing node. The same connection mechanism is used in section 4.2.8. The only alteration is that the device looking to connect is required to specify its type as peripheral.

The Peripheral Sensing System is configured to periodically gather all sensor data and send it to the Sensor Management System. Data is sent every 10 seconds, starting once the device is connected. An example of an outbound peripheral message can be seen in Listing 6

Listing 6: JSON Peripheral Sensor Readings Message Object

```
{
   "service":peripheral_sensor_values,
   "payload":{
      "peripheral_sensor_values":{
         "device_id":167469062838880,
         "touch":false,
         "light":14,
         "temperature":15,
         "time_stamp":2015-03-07 22:14:15
      }
}
```

4.4 RESTful Application Programming Interface

4.4.1 General Operation

To provide a remote communication interface for the Sensor Management System and the Android application, a representational state transfer(REST) architecture was implemented as the server side infrastructure for the project. The API runs on CS1, the departments web server. For storage, a MySQL database is used.

Its responsibilities include:

- Sensor data storage and retrieval
- Remote configuration of the Sensor Management System
- Remote camera operation
- Image and video storage
- Push notification registration ID storage and retrieval
- Aggregation of sensor data for graph presentation

REST was chosen over SOAP as it facilitates rapid application development. Each component of the project architecture is implemented in a different programming language. The interoperability of REST removes the complexity of platform specific API communication. All components can utilise a single RESTful interface over HTTP using JSON as the notation for messaging. This has the added benefit of making it instantly compatible with the current systems messaging format.

4.4.2 Request Management

Following the standard REST architectural style all communication is handled via HTTP requests. The content body of each message takes the form of a JSON encoded object. As described in section 4.2.7, Sensor Management System API Communication, an addition header value, called "service", is attached to each API request, specifying the desired service. The RESTful API simply reads this value, and manages the request payload accordingly.

4.4.3 Response Management

To reduce the number of API calls needed for operation, attached to the response of each request are a number of additional fields.

With all responses from the server to both the Sensor Management System, and the Android application, a status code for the completed service is included, to allow for reactive error handling. In keeping with the W3 RFC2616 HTTP/1.1 status code specification, the following status codes were used.

This method facilitates case based reactive error handling, and allows for easier debugging as the problem is automatically narrowed down to one of three possibilities. A timestamp is also included to for general debugging purposes. Depending on where the request originated, further additional fields may be included.

When the request originated from the Sensor Management System, a field called "camera_ operation" is added, with a numeric value attached, which represents the required camera operation. An example can be seen below in Listing 7.

Status	Status Code
Successful	200
Bad Request	400
Not Found	404
Server Error	500

Table 1: API Status and Corresponding Codes

Listing 7: JSON API Response to Sensor Management System

```
{camera_operation:"0",status_code:200, time_stamp:"2015-03-07 22:14:15"}
```

When the request originated from the Android application, the Sensor Management Systems public IP address is included in the response payload, which is needed went trying to access the remote stream from within the application. An example can be seen below in Listing 8.

Listing 8: JSON API Response to Android Application

```
{"status_code":200,"pi_public_ip":"89.101.115.84"}
```

4.4.4 Video Streaming and Image Capture

The API maintains a camera flag, stored persistently, which signifies the currently desired camera function to be executed, denoted by a numeric value. When the Android application makes a request for a remote stream, or image capture, the API updates this flag accordingly.

As described in section 4.4.3 above, when the Sensor Management System makes a request to the API, contained in the response is the camera_ operation field, populated by the value of the previously described camera flag.

In this way, camera functionality can be invoked by the user, without the need for additional HTTP requests. There is a problem with this implementation however. The invoking of the camera function is constrained by the frequency of communication from the Sensor Management System to the RESTful API. In situations when communication is taking place in large time intervals, the same latency is experienced by the user.

In order to provide remote video streaming, the Sensor Management System's public WAN IP address must be tracked at all times for use by the Android application when accessing the remote video stream. For every request sent from the Sensor Management System, the API records the public IP address persistently. As described in section 4.4.3 Response Management, the System's public IP address is passed back to the Android application with each response.

4.4.5 Data Storage

The API maintains 3 MySQL database tables, which take a simpler form than the original specification. The choice to simplify the schema was taken due to time constraints. The original design separating the sensor values table would limit the storage required, but also increase the underlying functionality needed to manage the additional tables. The final implementation accomplishes the same functionality, but at the loss of storage optimisation.

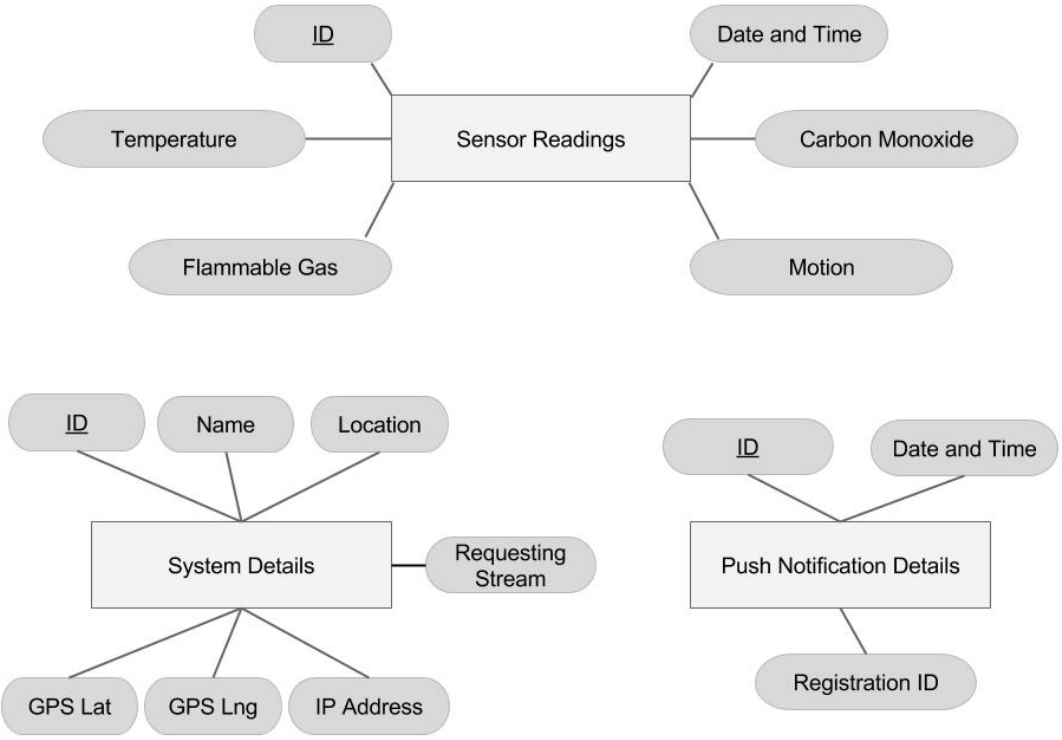

Figure 17: Entity Relationship Diagram for RESTful API Database Schema

4.4.6 Data Aggregation

To optimise data aggregation for graphical presentation the original design specified three database tables for sensor data storage. One table for the current day sensor values, one table for current day values aggregated by hour, and a final summary table for aggregated values for days previous. The aggregated values include maximums, minimums and averages.

Partitioning the data in this way would reduce the overall storage capacity required. However, actively performing this aggregation on the database at regular time intervals is necessary.

Due to time constraints, to achieve the needed functionality, a single table was implemented, as seen in the Figure 17. Aggregation is then performed on demand. This increases server load, and is not an optimal solution.

4.4.7 Push Notifications

When registration with the Google Cloud Messaging Service is complete, the resulting device registration identification number given back must be persistently stored for later use. As can be seen in the entity relational Figure 17 above, the Push Notification Details table is dedicated to storing these registration IDs. When an alert threshold is reached by the Sensor Management System, an API call is made for theses IDs, which are then based back and used to target the push notification to individual devices.

4.5 Android Application

4.5.1 Application Structure

There are three basic components of an Android application: activities, fragments and views. These form the basis for the Android view hierarchy.

The notion of an activity is a single, focused thing that the user can do. Activities create the window in which other components can be situated. They maintain a lifecycle, allowing the invocation of functionality depending on the current lifecycle stage.

A fragment is a self contained module which encapsulates a variable number of view components. It can only exist within an activity. It represents a behavior or a portion of a user interface. It too maintains a lifecycle, which is more extensive than that of an activity.

Views are the building blocks for a UI. Views can be static, like a TextView, or interactive, like a Button.

The choice was made to only uses a single activity within the application, and delegate functionality to individual fragments, which are subsequently embedded within this activity. In this way a clear separation of functionality can be maintained, and cohesion can be kept high. The benefit of this structure is that it facilitates more efficient application lifecycle and resource management, with single points of entry and exit. For example, in the eventuality of the application being closed, any active tasks such as API calls can be halted in a single location.

4.5.2 UI Development

Implementation of the Android application began with development of a general user interface structure.

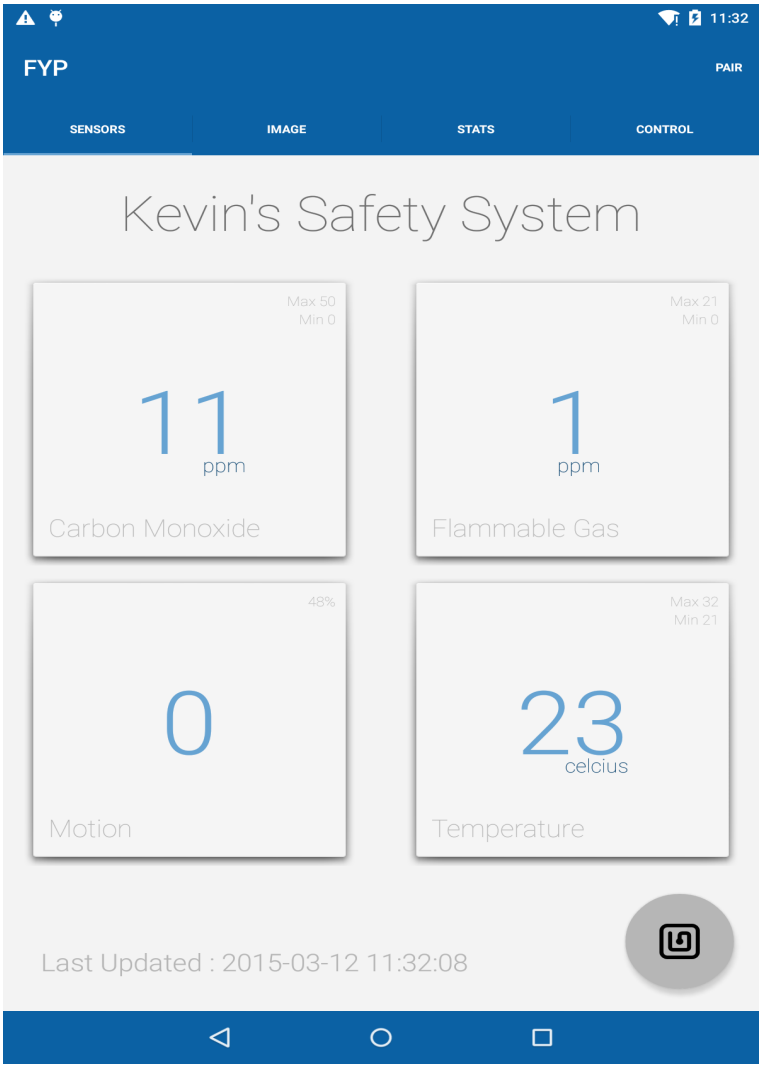

Figure 18: Sensor Data Screen within the Application

The UI follows a standard tabular design, which provides the user with an easy means of navigation. This can be seen in Figure 19 below. The benefits of this navigational pattern is the flexibility it provides for introducing new fragments, or alternatively removing them.

Figure 19: Navigation Bar within the Application

Four fragments exist within the application, corresponding to the tabs in Figure 19. They serve the following individual purposes:

1. Displaying of Sensor Readings, located under the Sensor tab
2. Camera Operation, located under the Image tab
3. Data graphing, located under the Stats tab

4. System configuration, located under the Control tab

4.5.3 Data Modelling and Management

In order to maximise performance and reduce code within the application, no persistent database storage is used. Instead, when possible computation is delegated to either the server side API, or the Sensor management system. This is to ensure that the application runs seamlessly, and is only required to perform small computational tasks. This allows for better management of limited resources to be designated to the application's user interface components, ensuring their fluid operation.

Data is modelled within the application using JSON, maintaining the same structure as used in the other subsystems of the project. A key component to achieving swift parsing and serialization of JSON objects is a library called Gson. Gson, also known as Google Gson, is an open source Java library to serialize and deserialize Java objects to, and from, JSON.

To facilitate the easy inflation of Java objects, first a standard java object must be defined, with the appropriate attributes, as seen in Listing 9 below. The names of each attribute must match that of the JSON object being deserialized.

Listing 9: Sensor object to be serialized

```
public class Sensor {
    protected String name;
    protected boolean is_active;
    protected int    priority;
    protected int    alert_threshold;
    protected int    probe_rate;
}
```

To serialize, you create a new GsonBuilder object, and pass the object to be serialized to the "toJson" method, as seen in Listing 10. This will return a JSON formatted string corresponding to the attributes present in the object, displayed in Listing 11.

Listing 10: Serialize object to string

```
String sensorJsonString = GsonBuilder().create().toJson(sensor);
```

Listing 11: Sensor string resulting from serialization

```
{
   name:"temperature",
   alert_threshold:50,
   is_active:true,
   priority:1,
   probe_rate:10
}
```

To deserialise the string, a new Gson object is created, and the fromJson method is invoked, passing in the JSON string and the Java class implementation to be populated. This returns a pure Java Sensor object, shown in Listing 12.

Listing 12: Deserialize string to sensor object

```
Sensor sensor = new Gson().fromJson(sensorJsonString, Sensor.class);
```

4.5.4 Data Presentation

Sensor value data is the first thing the user is presented with when opening the application. A GridView was used to achieve an extensible layout, with each sensor dedicated its own tile within the grid. Figure 20 shows a tile corresponding to the carbon monoxide sensor. The sensor name is located at the bottom of the tile. The maximum and minimum values for the current day are placed at the right hand top corner. Directly in the center is the latest value, and just below is the measurement, in this case parts per million.

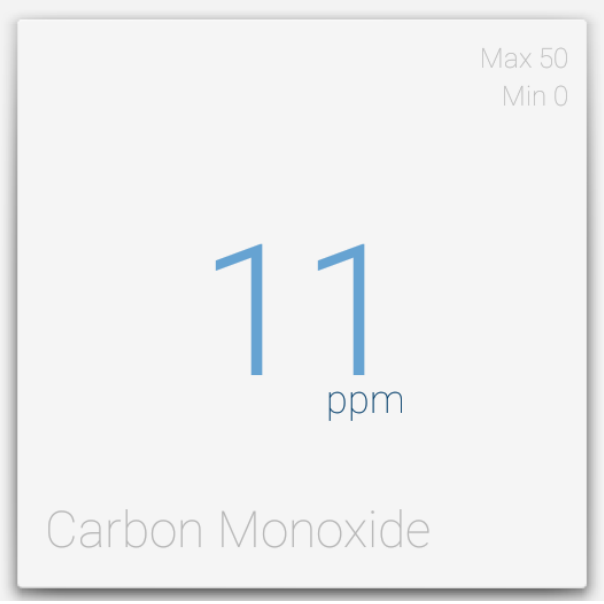

Figure 20: Sensor Tile

When a user defined sensor alerting threshold is reached, an additional icon appears on the tile corresponding to the alerting sensor. This can be seen in Figure 21 below. Note that this tile is for the infrared motion detector, and is binary valued, 1 denoting motion being detected. The value in the right top corner is the percentage of motion detected for the current day.

Figure 21: An Alerting Sensor Tile

Located above the sensor tiles is the name of the physical system the data is being displayed from. For the Sensor Management System this name is set by the user. Alternatively, when data is being displayed from a peripheral sensing node, it simply states the number of the node.

Figure 22: Input System Label

Figure 23: Input System Label for Peripheral Device

Underneath the sensor tiles in the bottom left corner is a text view which displays the date and time of the sensor readings, this can be seen in Figure 24. In the right corner is the button which toggles between the physical sensing systems.

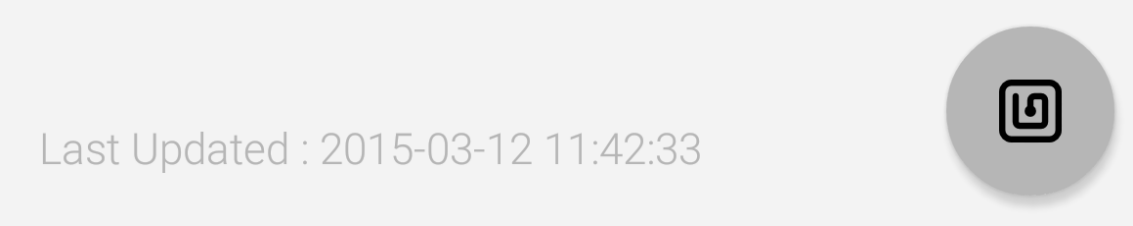

Figure 24: Last Updated Timestamp and System Toggle Button

Figure 25 below shows the result of toggling the sensing system, where the readings are being displayed for the Peripheral Sensing Node, which was running on the Intel Galileo.

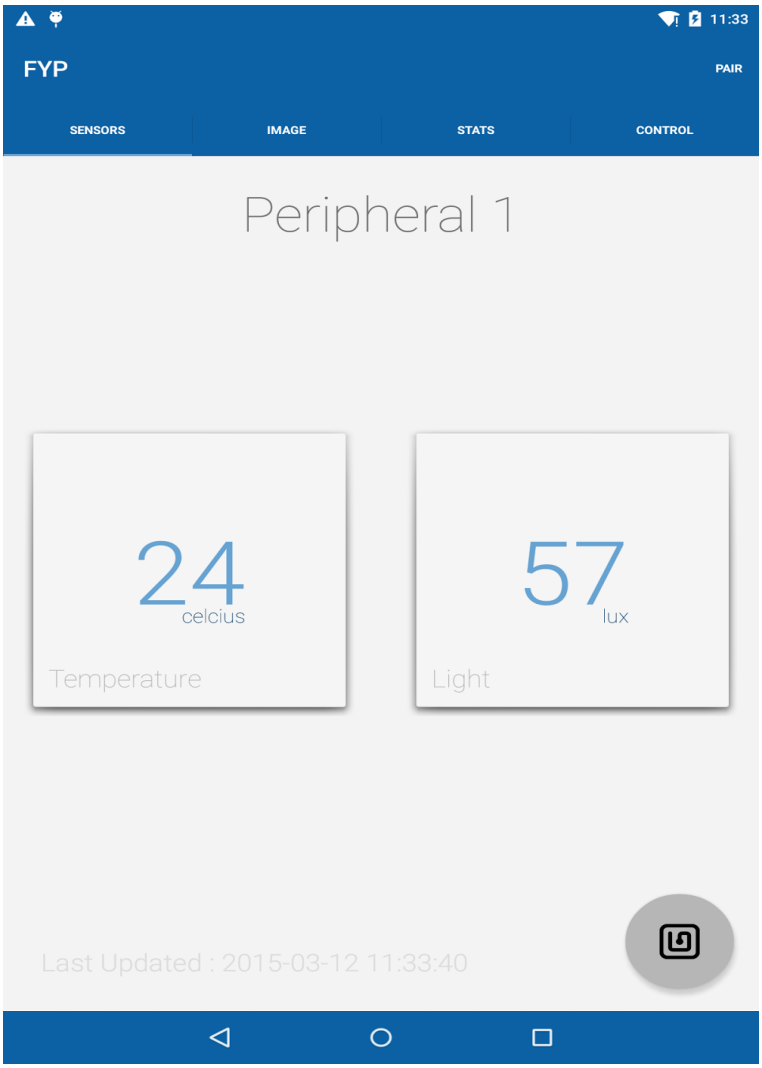

Figure 25: Peripheral Sensing Node Sensor Values

Each sensor tile has an on touch listener registered, so when a user touches a tile a dialog box appears. Context sensitive feedback corresponding to the current sensor value is presented to the user, and some information on the sensor itself. An example of which can be seen in Figure 26 below.

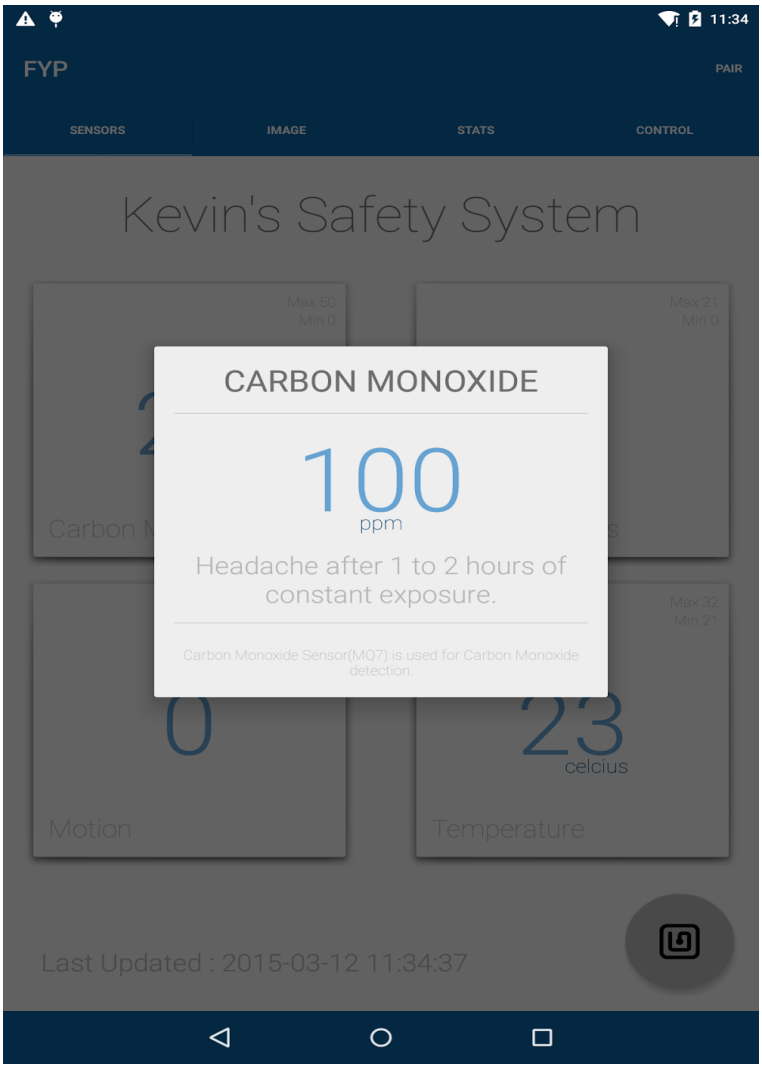

Figure 26: Context sensitive feedback for carbon monoxide reading of 100ppm

4.5.5 API Communication

All network activity in the application is handled by a single HTTP library called Volley, which is developed by Google. It simplifies and optimises request management. It provides automatic scheduling, multiple concurrent connections, caching for both text and image data, and easy cancellation of requests. Volley utilises a priority queue as its underlying data structure for request execution. Optimisation is handled through the use of background asynchronous tasks, ensuring the main UI thread is not blocked which can result in latency that is apparent to the user. Volley offers extensible objects for creating JSON HTTP requests, a key feature that was required for interacting with the server side RESTful API.

When developing the infrastructure for API communication, a singleton pattern was applied. This ensured that all network connectivity is coordinated by a single entity. Multiple benefits result from the use of this design pattern: classes which utilised API requests are kept simple as it is a noninvasive means of providing functionality, all currently executing requests can be cancelled simultaneously, which is useful when the application is halted by the user, and debugging is quarantined to a single location. Listing 13 demonstrates the invocation of a request for the latest sensor values.

Listing 13: Singleton API object method invocation

```
API.getInstance(getActivity()).requestSensorValues(sccssListener, errListener);
```

As mentioned in previous sections, in order to identify what service is being requested, an additional value is set in the header of the HTTP request. Volley allows a Map object to be added to the request specifying the header fields. The request itself is generic, using a placeholder object called APIResponse, which is populated when the response for the RESTful API is recieved. The APIResponse object can be seen in Listing 15. The request object possesses two event listeners, one for handling errors, and the other for successful responses. The APIResponse object is delivered to one of these listeners when the request is complete.

Listing 14: Java method API call for request latest sensor readings

```
public void requestSensorValues(Response.Listener<APIResponse> listener,
Response.ErrorListener errorListener){
       Utils.methodDebug(LOGTAG);

       //Create a Map for the HTTP headers
       Map<String, String> headers = new HashMap<String, String>();
       //Add the service for getting the latest sensor values
       headers.put(Constants.API_REQUEST_HEADER_SERVICE,
        Constants.API_REQUEST_SERVICE_GET_SENSOR_VALUES);

       //Create the request object
       GsonRequest<APIResponse> sensorValueRequest =
       new GsonRequest<APIResponse>(URL, APIResponse.class,
       headers, listener, errorListener);

       //Add request to queue to be executed
       addToQueue(sensorValueRequest);
   }
```

Listing 15: APIResponse class, used when parsing the RESTful API response

```
public class APIResponse {
    //MetaData Params
    public int status_code;
    public String pi_public_ip;

    //Requested Objects
    public CurrentSensorValuesFromServer sensor_values;

}
```

4.5.6 Direct Communication

To provide direct communication from the Sensor Management System to the application, the Android device must be connected via the access point which is being administered by the Raspberry Pi. When pairing with the system, the application creates a multicast socket and joins the group, which the Sensor Management System is perpetually a member of. A pairing session object, encoded in JSON, is sent to the multicast channel from the application. The session object identifies the device and provides some general monitoring information. An example of this JSON session object can be seen in Listing 5 in Section 4.2.8.

On receiving this session object, the Sensor Management System performs the necessary pairing operations and sends back a status message, either verifying a successful connection, or failure.

Natively on Android multicast networking is switched off to reduce battery consumption. To perform multicast operations, a lock must be acquired from the operating system, seen in Listing 16. Because of this limitation, the only multicast message sent from the application is the pairing session object, and after sending the multicast socket is closed, and the lock released. All further communication is handled via TCP sockets. This has the additional benefit of reliability.

Listing 16: Multicast lock acquisition

```
//Get the WifiManager Server from the OS
WifiManager wifiManager =
(WifiManager) context.getSystemService(Context.WIFI_SERVICE);
//Get the lock object from the WifiManager
WifiManager.MulticastLock lock = wifiManager.createMulticastLock(LOGTAG);
//Aquire the lock and begin multicast networking
lock.acquire();
```

The singleton design pattern was applied to the object that was delegated the responsibility of communication between the application and the Sensor Management System. The object was entitled SocketManager, as its main duty is the orchestration of TCP/IP sockets. By maintaining a single point of interaction, the management of communication is straightforward. Overall application complexity is reduced as messages can be passed from any code location, without having the implementational overhead of passing the SocketManager object to each class which requires its use.

Below in Listing 17 is an example of a direct message to the Sensor Management System, in this case a new configuration is being sent. We simple utilise a static method, getInstance, on the SocketManager object, passing to it the Android application context, this returns the singleton object which can subsequently be used to send a direct message to the Sensor Management System. The sendMsgToPi method simply takes a JSON encoded string and sends it to the Raspberry Pi.

Listing 17: Singleton SocketManager object send message method invocation

```
SocketManager.getInstance(getAppContext()).sndMsgToPi(Utils.toJson(newConfig);
```

4.5.7 Push Notifications

A number of steps are required to implement push notification alerting. Figure 27 shows the interaction between the 4 entities involved.

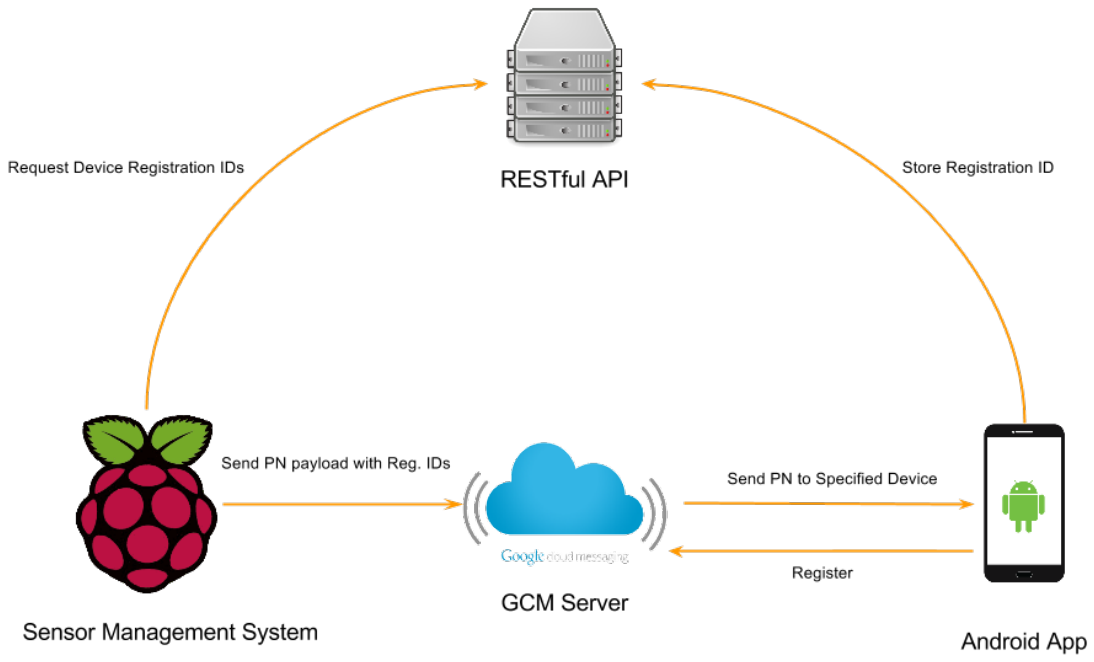

Figure 27: Push notification interaction doagram

The steps involved are as follows:

1. Create a new project in the Google Cloud Developer Console
2. Authorise the project to grant it access to the Google Cloud Messaging(GCM) Service
3. Authorise the project to grant it access to the Google Cloud Messaging(GCM) Service
4. Perform in app registration of the device to the GCM server, which generated a unique device identification number. This must be stored persistently by the RESTful API and is used to target notifications to the specific device
5. Extend a IntentService object within the app which will act as a daemon for receiving the notification payload from the GCM service
6. Extend an Wakeful Broadcast Receiver object within the app which will wake the device up if its on standby mode, and display the notification to the user
7. And the final step is to implement a program that performs the push notification request, via HTTP, to the GCM server, specifying the registration ids and the payload to be pushed to devices.

After an alert threshold has been broken and a push notification has been sent from the Sensor Management System to the Android Application, the resulting event of receiving the notification can be seen in the Figure 28 below. A white popup box appears at the very top of the screen. This box will appear regardless of the current foreground application.

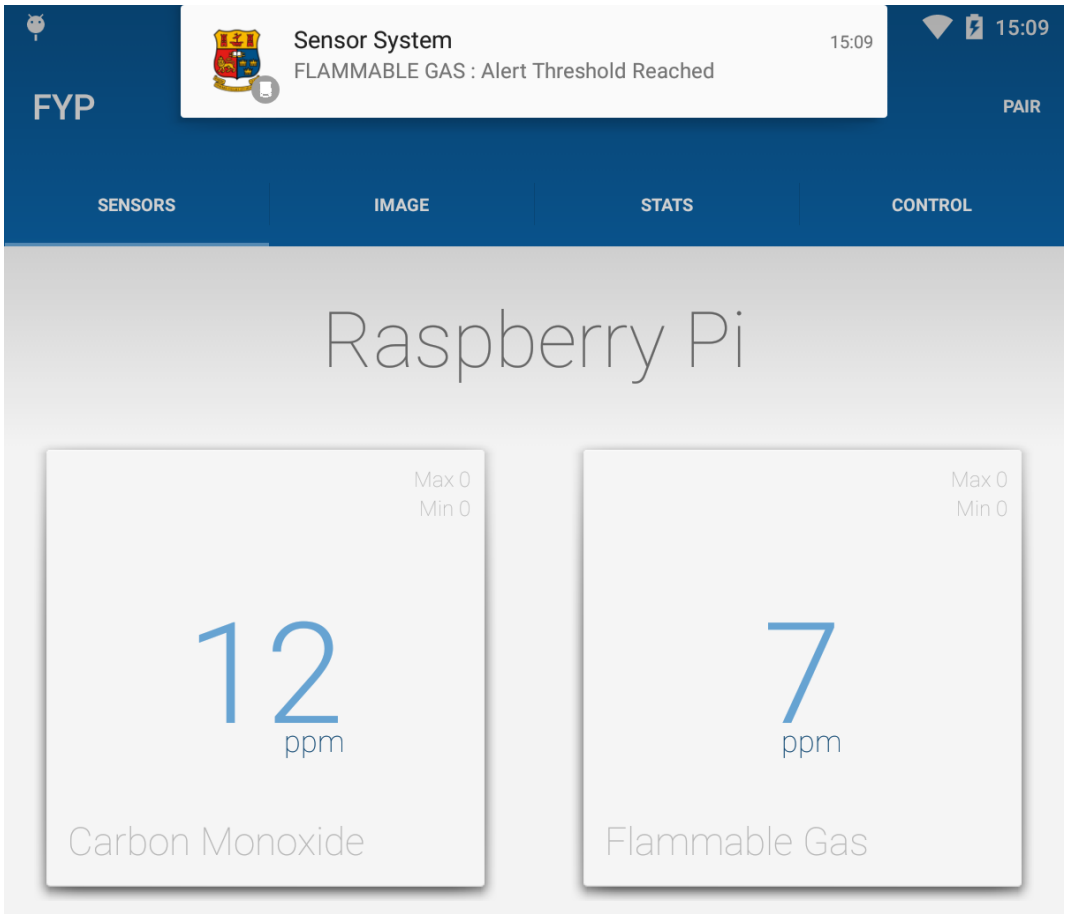

Figure 28: Push notification being displayed

Android has a mechanism, called PendingIntent, for opening an application by touching a push notification. This allows users to navigate directly to the application which has performed the alerting. When the push notification is above is clicked on, the application will open and display some context sensitive help regarding the current sensor value. This can be seen in the diagram below.

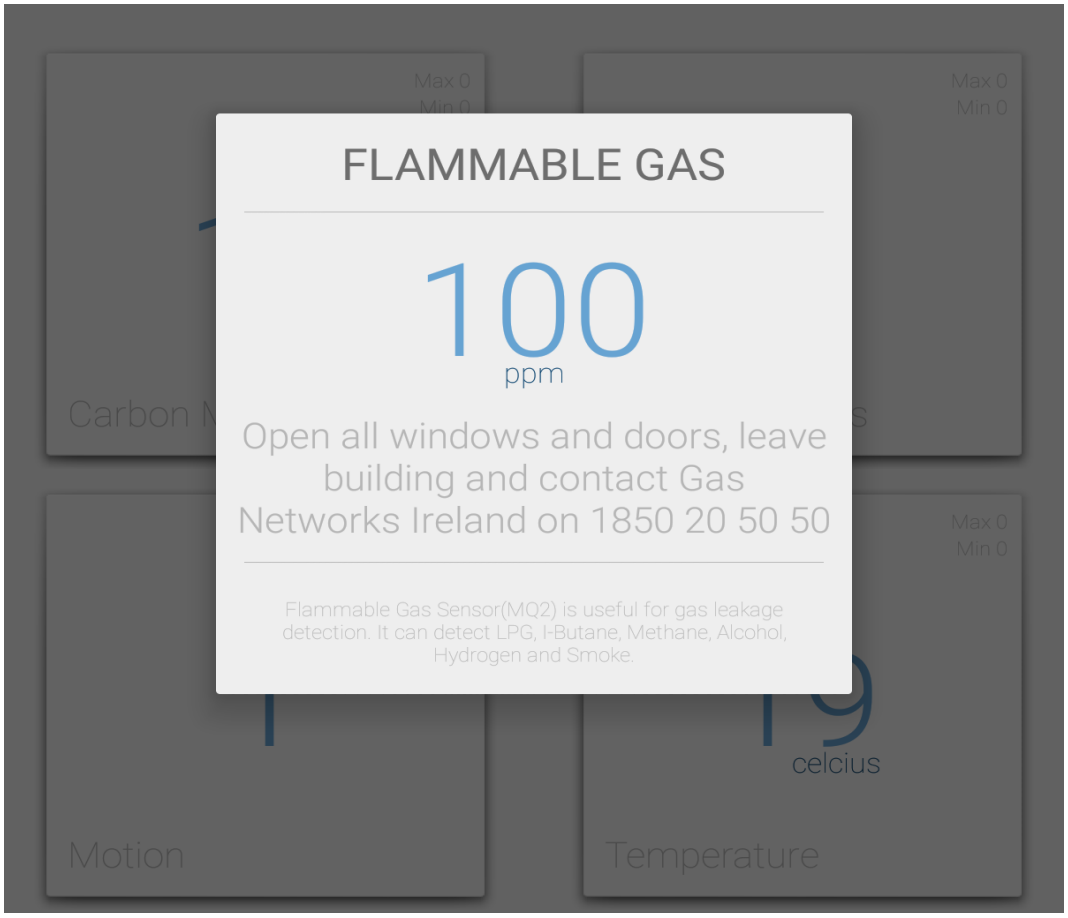

Figure 29: Conrext sensitive help after touching push notification

4.5.8 Context Sensitive Help

Little indication of safety conditions is provided by the sensor readings alone. In order to provide them with genuine feedback on the current environment and their own safety, a small dataset of sensor specific safety metrics was gathered from the internet. The entire dataset can be seen in Appendix B, Section 8.

4.5.9 Graphing

Graphing is provided by an open source library call MPAndroidChart [4]. This library provides a comprehensive range of charts, and a large amount of functionality, including touch scaling, real time graphing and animations. All graphing is designated to the Stats fragment within the application.

The application requests the graphing data from either the RESTful API, or directly from the Sensor Management System depending on the method of communication being utilized. All data aggregation is performed outside of the application. When a request for graphing data is executed, the respective RDBMS handles the aggregation via SQL query. This facilitates the generation of average, max and minimum values for each column.

[4]https://github.com/PhilJay/MPAndroidChart

Line charts were chosen as the most effective way of conveying the data to the user. All graphs are interactive, and allow the user to zoom in or out on specific areas.

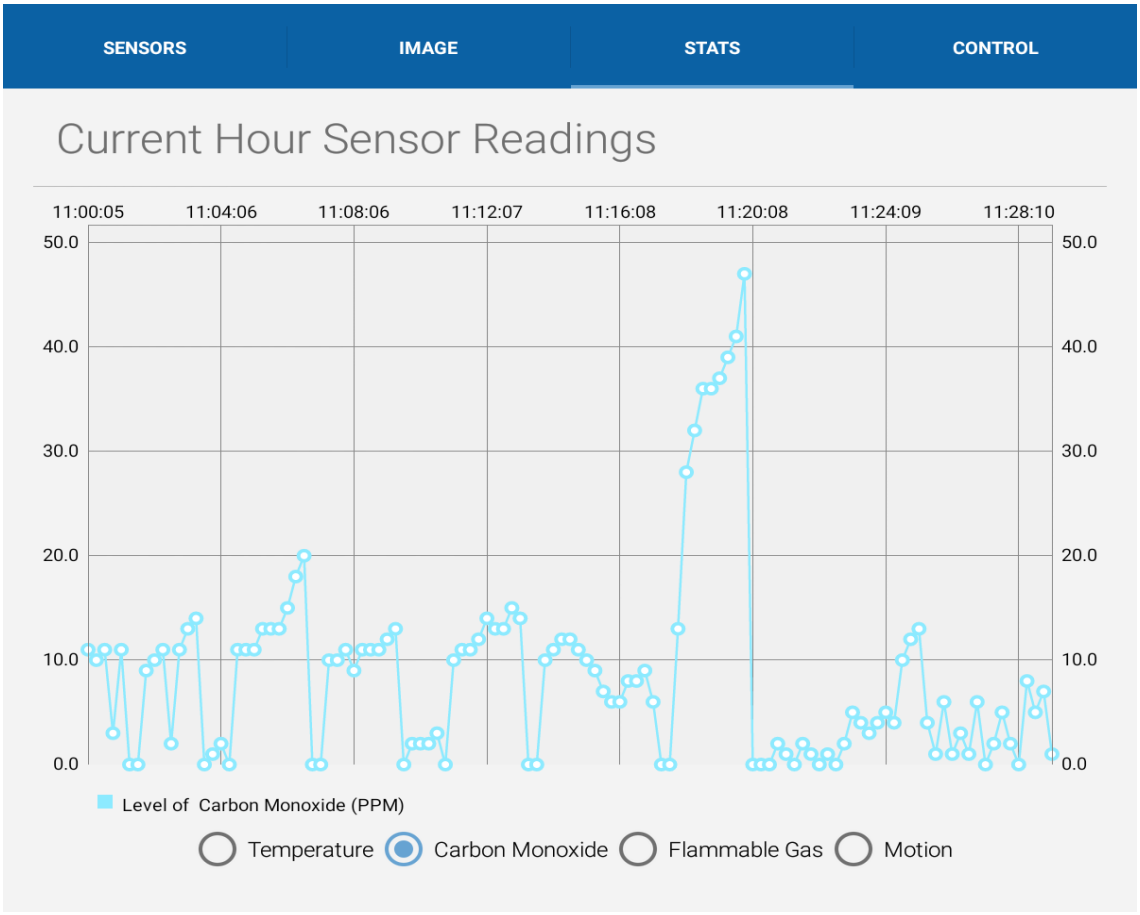

Figure 30: Current hour sensor values line chart

The Current Hour Sensor Readings chart graphs the current hour sensor values. The Y-axis denoted the sensor value, while the X-axis marks time.

The Current Day Summary chart plots aggregated values for each hour the system has been active that day.

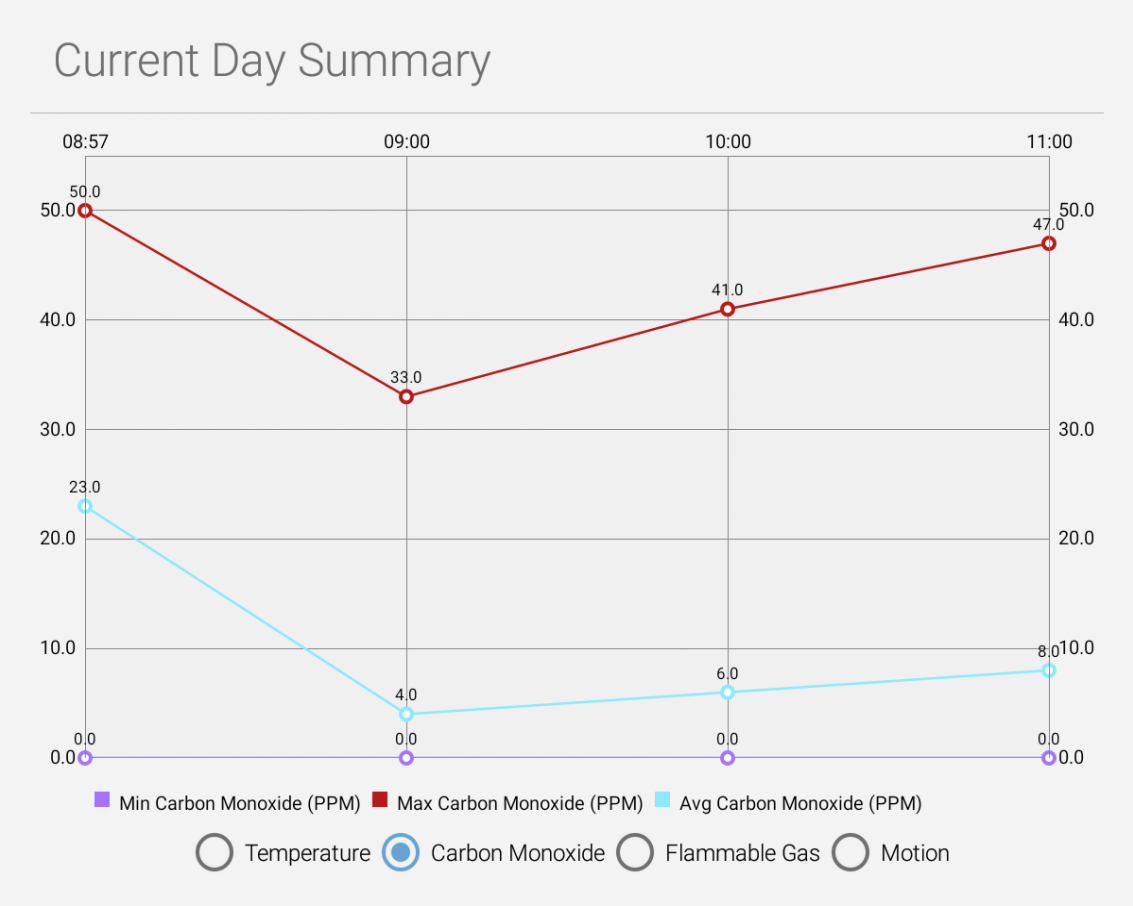

Figure 31: Current day sensor values aggregated by hour line chart

The Complete Summary chart graphs aggregated values for each day the system has been active.

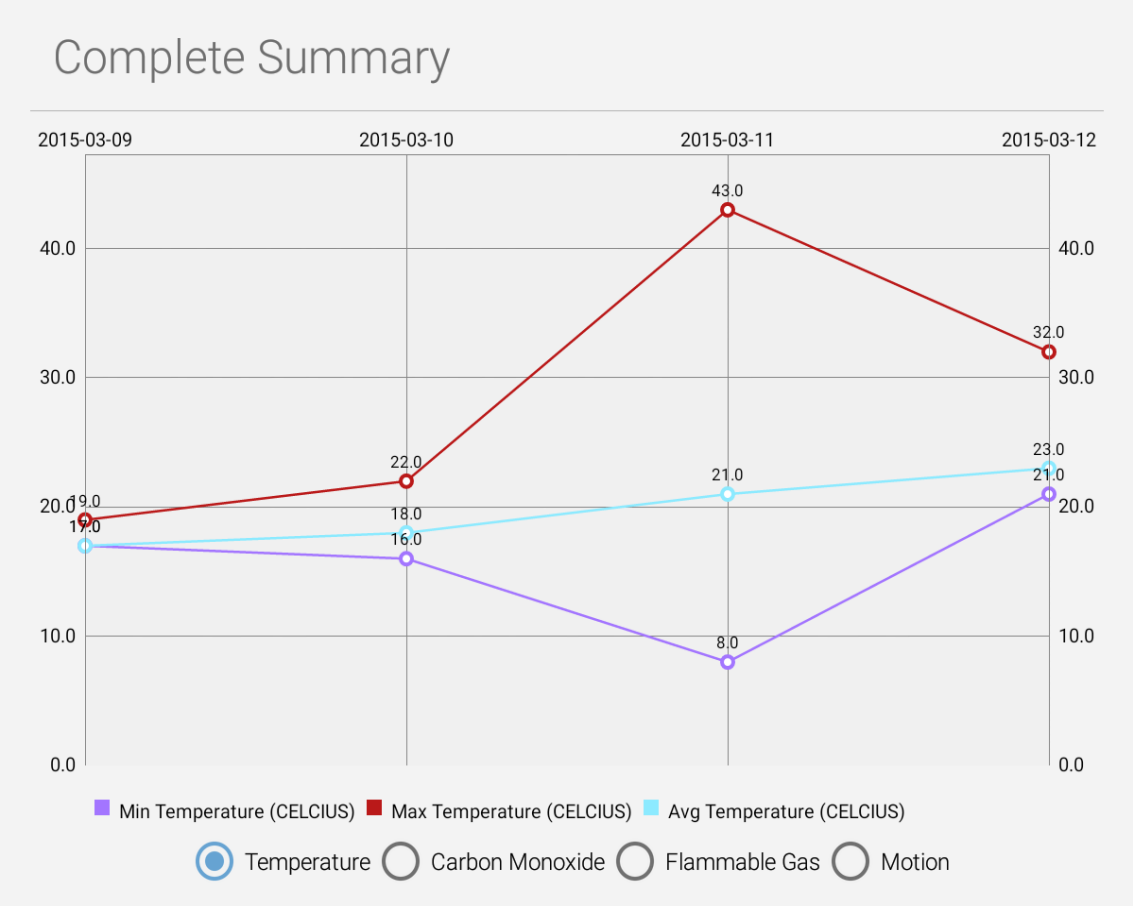

Figure 32: Summary of sensor values aggregated by day

The motion detector sensor data is displayed in the Current Day Summary and the Complete Summary charts as a percentage. This acts as an activity measurement, providing the user with the level of activity for a given time. This data could be used for home automation. An example of motion graphed as a precentage can be seen in Figure 33 below.

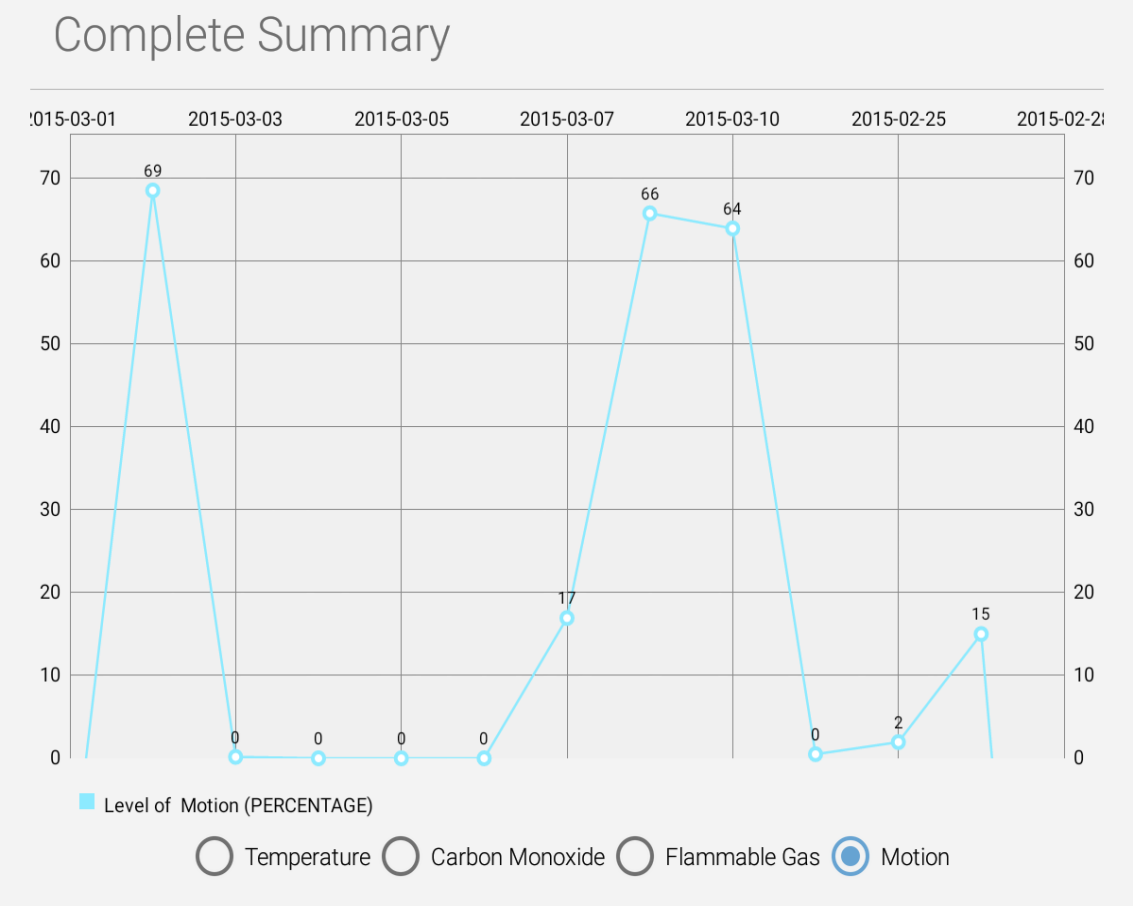

Figure 33: Motion graphed as a precentage

4.5.10 Camera Control and Video Streaming

The application provides image capture, streaming, and a gallery of images and videos previously captured. As describe in Section 4.4.4, RESTful API Video Streaming and Image Capture, image capture and streaming is facilitated by means of a HTTP request to the RESTful API where a flag is set, and delivered to the Sensor Management System, where the desired operation is executed.

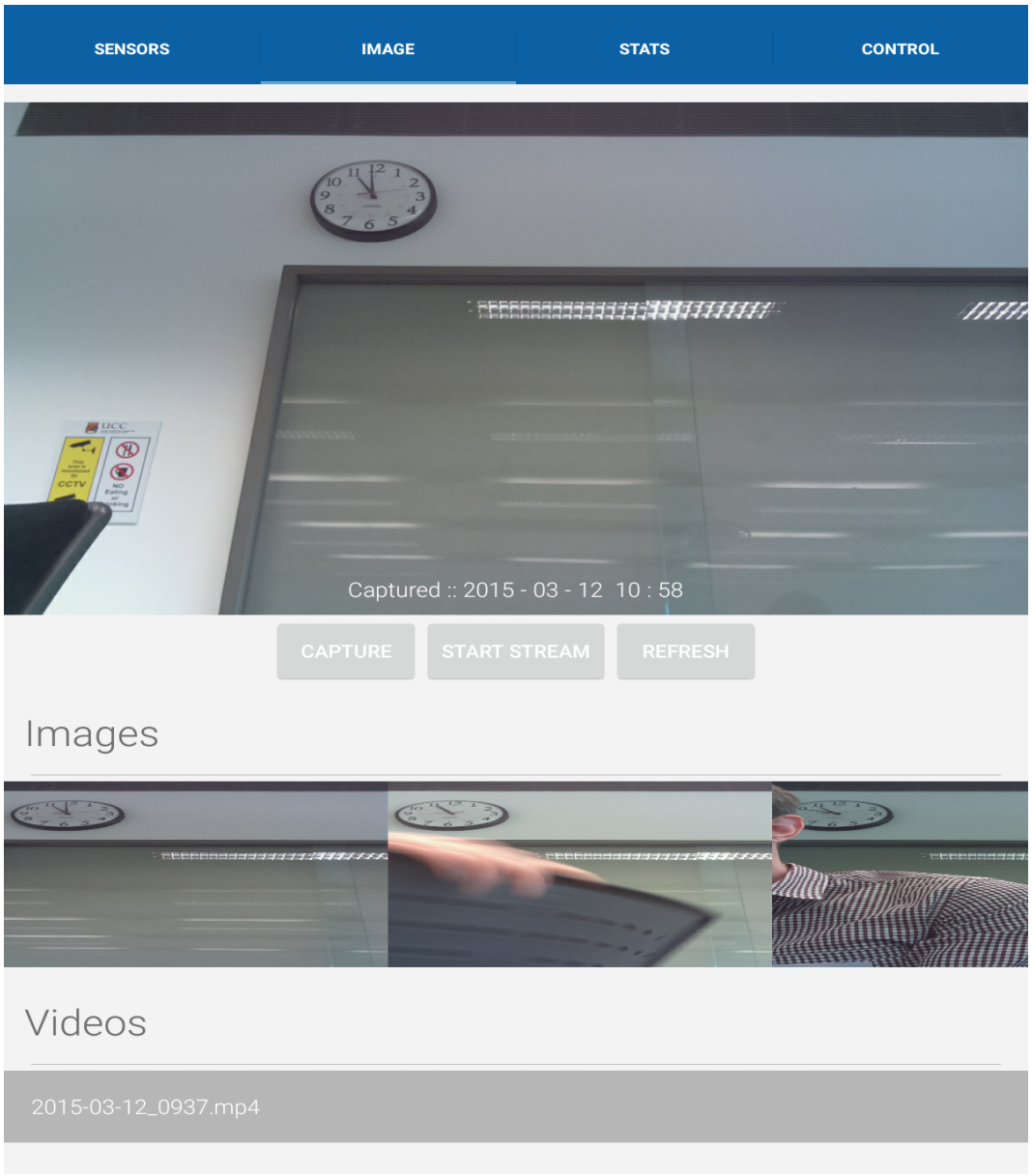

Figure 34: Image fragment for controling camera operation

When communicated via the RESTFUL API, all images and videos are stored server side and requested when the the user navigated to the Image fragment. All requested assets are cached, which is handled by the Volley Networking library.

When communicated directly via the AP, the application connects to the SFTP server that's running as a daemon process on the Raspberry Pi. Image and video files can then be transferred programmatically directly to the devices internal storage, or more specifically the applications private storage directory. When the user presses the refresh button, which can be seen in the Figure 34 above, an SFTP connection is made, and all new image and video files are transferred to the device. These file are then simply read from internal storage each time the user navigates to the Image fragment within the application.

Streaming and video playback are handled by the native Android VideoView API. A uniform

resource identifier(URI) is passed to the VideoView, and in the case of a static video, depending on the communication medium, the URI is either a URL pointing to the video stored remotely on the RESTful API, or when directly communicating, a file location within internal device storage. The Android VideoView has the functionality to download and play a video file simultaneously.

When a request is made for a remote video stream, the URI given to the VideoView is the URL constructed from the Sensor Management System's public IP address, and a predefined port number. The application polls this URL until the stream starts.

4.5.11 System Configuration

As described in the Section 4.2.4, Sensor Management System Configuration, the JSON object notation is used to configure the system. The Control fragment within the application provides the user with an interface for altering configurations. It was implemented as a form, with each section titled by the underlying system component it configures.

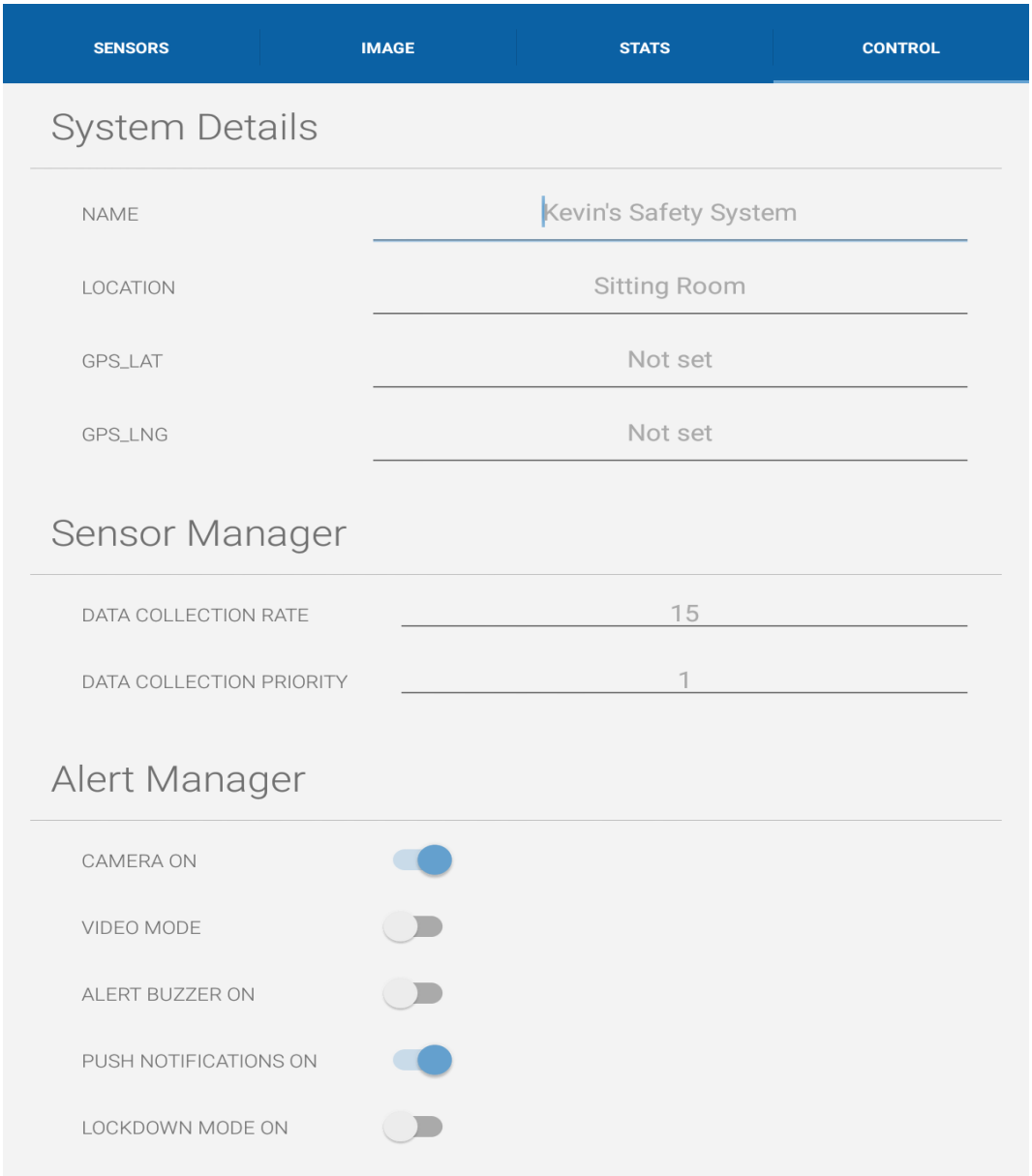

Figure 35: System Details, Sensor Manager, and Alert Manager configuration form fields

The System Details section, in Figure 35 above, allows the user to name the Sensor Management System, set its location, and GPS latitude and longitude values.

The Sensor Manager section, in Figure 35 above, handles the time interval, in seconds, at which the sensor data is written out to persistent storage. This affects the granularity of the data being provided to the charts in the Stats fragment. A priority can also be set on the thread which executes the sensor data storage.

The Alert Manager section, in Figure 35 above, provides the ability to turn on or off the camera, alert buzzer, push notifications, and lockdown mode. When the lockdown mode is active, if motion is detected, the camera will be activated and a push notification will be sent to the user. Switching from the default of image capture to video capture is also facilitated.

Figure 36: API Manger and Access Point Manager configuration form fields

The API Manager section, in Figure 36 above, configures how often sensor values, and camera files are uploaded to the RESTful API. It also allows the user to set how often the system should check for new configurations.

The Access Point Manager section, in Figure 36 above, sets the rate at which sensor values are sent to the application when directly communicating.

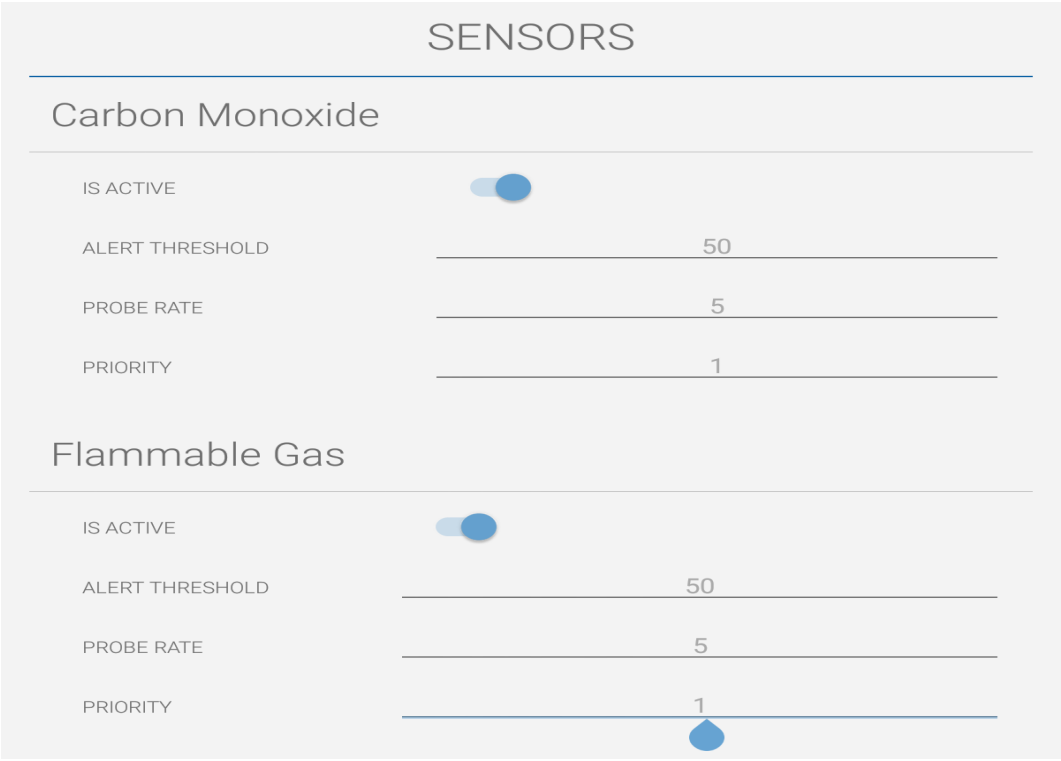

Figure 37: Sensor configuration fields

Each sensor connected to the Sensor Management System is configurable by the user, including its alerting threshold, probe rate and priority. Sensors can also be turned on or off. This can be seen in Figures 37 and 38.

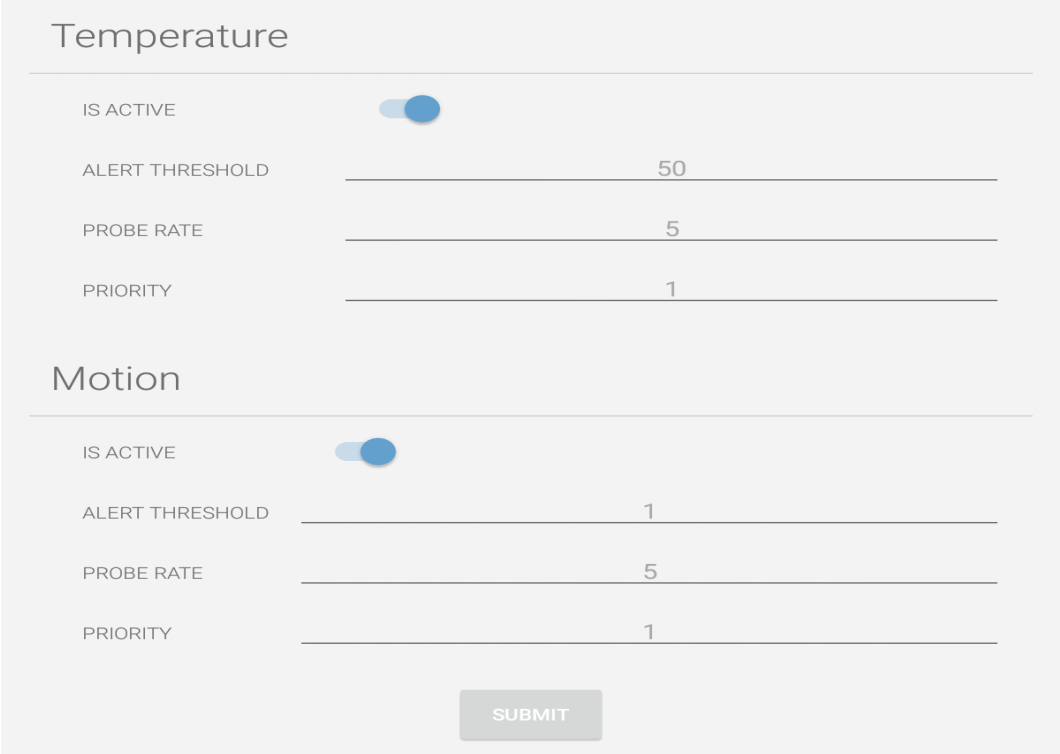

Figure 38: Sensor configuration fields

The form's submit button can be seen in Figure 38. This will execute the collection of form field data, and will format it as JSON. The subsequent action is dependant on the communication medium. If directly communicating, the configuration will be delivered straight to the Sensor Management System, and it will be reconfigured within 1 second. When communicating via the RESTful API, the new config file will be uploaded, and requested by the Sensor Management System at the time interval set by the API Manager in Figure 36.

5 Testing and Evaluation

5.1 Continuous Integration Testing

Continuous integration(CI) was followed throughout the development phase of the project. The Git code version control system was utilised, which allows the creation and management of cloud based code repositories.

To perform CI, a local working copy of the code repository is maintained, this is then merged with the cloud based remote repository once changes have be implemented, tested and verified. In this way, a stable version of the software is always available and backed-up. The main benefit of this code versioning approach is that changes can be reverted. For example, if a problem is found in the system, the entire repository can be reverted to an earlier working version. Additionally, for every change committed to the remote repository, a description of the alterations can be included, providing a development log, an invaluable resource when when reviewing the codebase.

5.2 Verification

Verification testing took place throughout the development phase of the project. This is the process of ensuring the system is built according to the requirements and design specifications. I worked closely with the functional and nonfunctional requirement specification, trying to keep the system aligned with the core features previously defined. The following features present in the initial system specification were omitted from the implementation.

Due to time constraints, no device authorization mechanism was implemented. This was envisaged to provide a means of Android device privilege management. In the current implementation, all devices have total control over the system, with no central administrator. This in practise would not be a viable solution, as anyone, trusted or untrusted, could alter the operation of the Sensor Management System.

The original specification proposed the ability for easy inclusion and integration of any desired sensor module, this was not achieved. The idea I had devised was to allow the user to connect a sensor directly to the Raspberry Pi, specifying its name, type, and GPIO pin it was connected to, and then the system would dynamically adapt to manage this additional sensor. While this is feasible in terms of sensor management and integration in the circuit, an architectural limitation stopped the implementation of this type of dynamic inclusion. The limitation faced was with the data layer, specifically the non-dynamic schema of Relational Database Management Systems. While it is possible to programmatically alter the database and add or remove columns, the complexity of the data management layer would have increased drastically, limiting my development time elsewhere. An alternative to overcome this, would be to switch to a NoSQL database. The dynamic schema of a NoSQL database would easily facilitate the addition or removal of sensors.

Other than these two shortfalls of the implementation, the specification was largely achieved.

5.3 Operational Timing

There are a multitude of operations being performed asynchronously in Sensor Management System. Due to the Raspberry Pi having only a single core, multithreading has not provided much improvement in timing, possibility even introducing a negative impact on system performance. Testing functionality execution times is difficult as race conditions are commonplace. To conduct operational timing testing, I simply altered the configuration file setting all timing intervals to 10 seconds, and recorded the debug printing logs of the system, deducing the mean task execution times from the differences between the configured time interval and the actual time of execution.

It was observed that the actual task execution timing varied by up to an additional 2 seconds on all operations. This variability in configured time interval compared to actual execution time is likely down to a race condition for processor utilisation. Table 7 displays core system operation timings in seconds.

Operation	Mean Execution Time in Seconds
Querying single sensor	0.2
Query all sensors and write values to persistent storage	1
Push notifications from the time of sending to receival	10
Uploading sensor data to the RESTful API	0.5
Downloading data from the RESTful API	0.6
Connecting directly via Android application	4
Direct data transfer	0.5

Table 2: Operation mean execution times in seconds

5.4 Performance and Stress Testing

The system holds up reasonably well under moderate loads, however this is largely depending on configured operational timing intervals. This was a difficult area to perform any conclusive tests in as the system's performance is closely coupled with the limited hardware resources of the Raspberry Pi. With the added overhead of background services running on the Raspberry Pi, such as the SFTP server and the Access Point with DHCP server, the Sensor Management System has limited computational space to operate in.

I configured the Sensor Management System to perform all operations at a timing interval of 1 second, the minimum interval allowed to be set by the Android UI. I observed that database writes became slow and backlogged as system execution progressed over time. After 20 minutes of system operation the delay amounted to between 10 and 15 seconds. API calls became slower to execute, and as a result, data was transferred to the RESTful API at delays of 5 to 10 seconds.

The Sensor Management System was robust enough to continue operation, performing alerting when needed, and the camera remained fully operational. However all operations had a latency which only grew as the system execution progressed over time. I attempted to further reduce the time interval to 0.5 seconds, but the same latency conditions were observed, but were increasing at a more rapid pace. It was found that the Sensor Management System operates optimally when timing intervals are between 5 and 10 seconds. Preferably, system operations should be scheduled at varying timing intervals, dispersing computational load more gradually.

As the Peripheral Sensor System is simplistic in design and operation, it can handle a very high volume of operations a second. Ive tested the send rate of sensor values from the Peripheral Sensing Node to the Sensor Management System up to a timing interval of 0.3 seconds, and it operates without any noticeable jittering. This really displays the level of computational power a Complex Instruction Set architecture has over a Reduced Instruction Set.

For the Android application, I performed all development and testing on an Asus Nexus 7 tablet, which has a quad core processor and 2 GBs of RAM. Inbuilt in Android Studio is a memory usage visualization tool. The maximum RAM usage I experienced when testing the application was 80

MBs. The average was around 50 MBs. In comparison, the default clock application which is packaged with the Android OS has an average RAM usage of 21 MBs. On average the application performs moderately, with minimal latency detectable. However, when graphing sensor values, a drop in performance is sometimes experienced, dependent on the number of sensor readings to be plotted.

5.5 Failures

In my opinion the most significant failure in the project was not being able to successfully implement the Wifi Direct infrastructure originally proposed. This would have allowed peer-2-peer communication directly with the system, while simultaneously maintaining a normal connection with local networking infrastructure. The overall functionality of the system would have been largely the same, but in terms of the Android application operation, Wifi Direct would have provided a more elegant solution with less scope for implementation errors. In general, low level networking on mobile devices is difficult to implement optimally. Instead of being able to use Android's comprehensive Wifi Direct APIs, I had to implement all wireless networking functionality manually, which increased the development phase of the project considerably. The main repercussion of meant that the Android device has to be connect to the Raspberry Pi's access point in order to communicate directly. This forces the user off their own wifi connection and is a poor solution in terms of user experience.

Something I took away from this failure is that new technologies should only be included in a system's architecture after a thorough feasibility analysis has taken place. Only after a concrete prototype has been implemented, should the technology be adopted. I made the mistake of basing a central component on an emerging technology which has not found mainstream usage yet. While Wifi Direct promises the functionality of Bluetooth, but at much greater speeds, without developer community support its inclusion in projects will remain a risk.

6 Conclusions

6.1 Potential Improvements and Going Further

A key aspect not addressed is the intelligent use of the data to automate other systems present in the home. For example, as the infrared motion detection sensor can anonymously monitor activity, the data could be used along with the temperature to schedule central heating for times when motion is highest, and temperature is low. Similarly, the system could switch off alerting services if someone is cooking, and then switch alerting back on once they have vacated the room.

The configurable nature of the system allows a lot of flexibility, but for the end user, this would result in more confusion than benefit. A better solution would be to introduce an intelligent agent within the system to automate its configuration to best suit its environment. This could be achieved by using aggregated sensor values to set alerting thresholds.

Introducing a duel alert monitoring system to improve safety protection. This would add to the original mechanism for sensor value alerting by introducing monitoring server side. This could proactively alert the user if early warning signs of danger occur. A prediction model could be learned from previously experienced events, and used to deduce the likelihood of potential dangers from current sensor values.

Changing the direct communication interface from Wifi to Bluetooth would be an essential alteration to the current implementation. This would improve the system's overall usability considerably, no longer requiring the user to connect directly to the system's access point.

Security doesnt feature in the current implementation. I chose to focus on developing the core

functionality of the system, and did not get to spend development time on improving security. Going forward, security would have to take a central role.

The polling mechanism the Sensor Management System currently utilises for API communication is not very optimal. A better solution would be to implement a distributed publish-subscribe architecture server side, that could push out events as the occur. To accomplish this, the Raspberry Pi would be required to maintain a web server, something the current implementation could not withstand. However, with the latest Raspberry Pi model 2, this would be feasible.

Migrating to a NoSQL database system would improve system flexibility, and allow the dynamic inclusion of sensors. Moving forward, this is an essential change, providing the infrastructure to allow users to plug in new sensors at will.

7 Appendix A

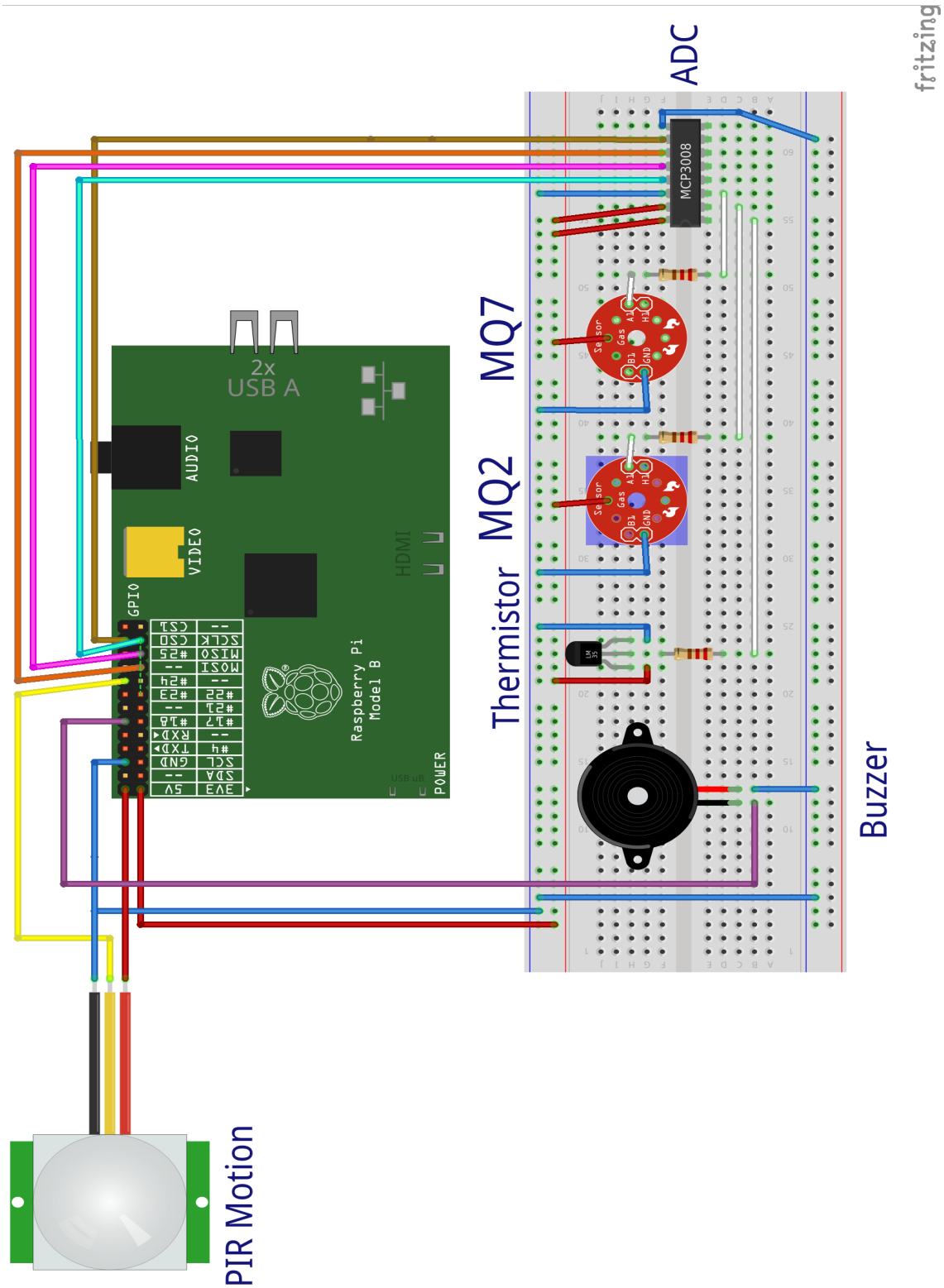

Figure 39: Wiring diagram for Raspberry Pi running the Sensor Management System

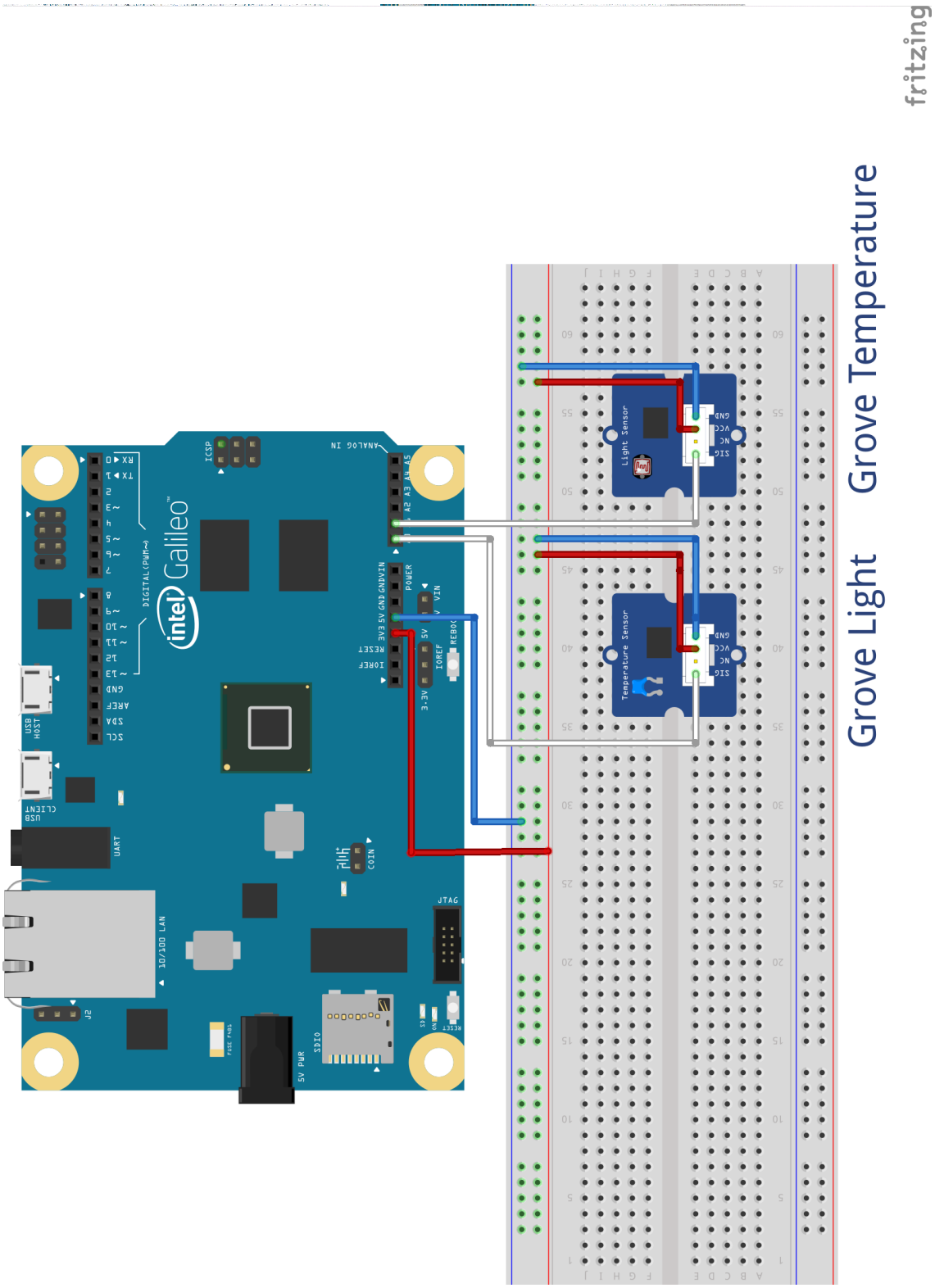

Figure 40: Wiring diagram for Intel Galileo running the Peripheral Sensor System

8 Appendix B

Carbon monoxide context sensitive help table, taken from http://www.detectcarbonmonoxide.com/co-health-risks/.

Feedback Text	**Level in Pars Per Million(PPM)**
Safe Level	0
Maximum legal level for the work place	25
Headache and Dizziness within 6 to 8 hours of constant exposure	35
Headache after 1 to 2 hours of constant exposure	100
Dizziness, Nausea, Fatigue, Headache after 2 to 3 hours of constant exposure	200
Nausea and Headache after 1 to 2 hours, life threatening in 3 hours.	400
Dizziness, Nausea and Headache after 45 minutes, unconsciousness after 2 hours	800
Dizziness, Nausea and Headache after 20 minutes, unconsciousness and possible death after 2 hours	1600
Dizziness, Nausea and Headache after 5 to 10 minutes, unconsciousness and possible death after 15 minutes	3200
Dizziness and Headache after 1 to 2 minutes, unconsciousness and possible death between 2 and 15 minutes	6400
Danger of death in 1 to 3 minutes	12800

Table 3: Carbon monoxide context sensitive help table

Flammable gas context sensitive help table, http://inspectapedia.com/sickhouse/Gas_ Exposure_ Limits.php

Feedback Text	**Level in Pars Per Million(PPM)**
Safe Level	0
Check gas appliances, potential leakage. Do not light any open flames.	25
Check gas appliances, potential leakage. Open all windows. Do not light any open flames	50
Open all windows and doors, leave building and contact Gas Networks Ireland on 1850 20 50 50	100

Table 4: Flammable gas context sensitive help table

Temperature context sensitive help table, taken from http://en.wikipedia.org/wiki/Orders_ of_ magnitude_ (temperature)

Feedback Text	**Level in Celsius**
Sub-zero, greatly increased risk of Pneumonia	Below 0
Close to Freezing, increased risk of Pneumonia	0
Below recommended room temperature, turn on heating to decrease risk of Pneumonia	10
Regular room temperature	20
Above recommend average room temperature	30
Possible fire, perform safety checks	40
Possible fire, time to exit the building	50

Table 5: Temperature context sensitive help table

Motion context sensitive help table.

Feedback Text	**Boolean**
No motion currently being detected	0
Motion currently being detected!	1

Table 6: Motion context sensitive help table

Light Ccntext sensitive help table, taken from http://www.engineeringtoolbox.com/light level-rooms d_ 708.html

Feedback Text	**Level in LUX**
Typical level for a moonless night	Below 0
Typical level for a full moon on a clear night	1
Typical level around twilight period	5
Typical level family living room lights	50
Typical level office building hallways/bath rooms	80
Typical level for a dark overcast day	100
Typical level for office lighting	300
Typical level for sunset or sunrise on a clear day	400
Typical level for overcast day, or TV studio lighting	1000
Typical level for full daylight	10000
Typical level for direct sunlight	30000

Table 7: Light context sensitive help table