Automated Safety and Security System for the Home

Kevin Patrick Murphy

March 24, 2015

Abstract

Contents

Acknowledgments

1 Introduction

1.1 Project Goals

2 Problem Analysis

2.1 Literature Review

2.1.1 Internet of Things

2.1.2 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.

2.1.3 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

2.1.4 Embedded Computing

2.1.4.1 Raspberry Pi

2.1.4.2 Intel Galileo

2.1.5 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.

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.

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.

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

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 projects 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.

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 smart phone application
- Peripheral sensing node running on an Intel Galileo

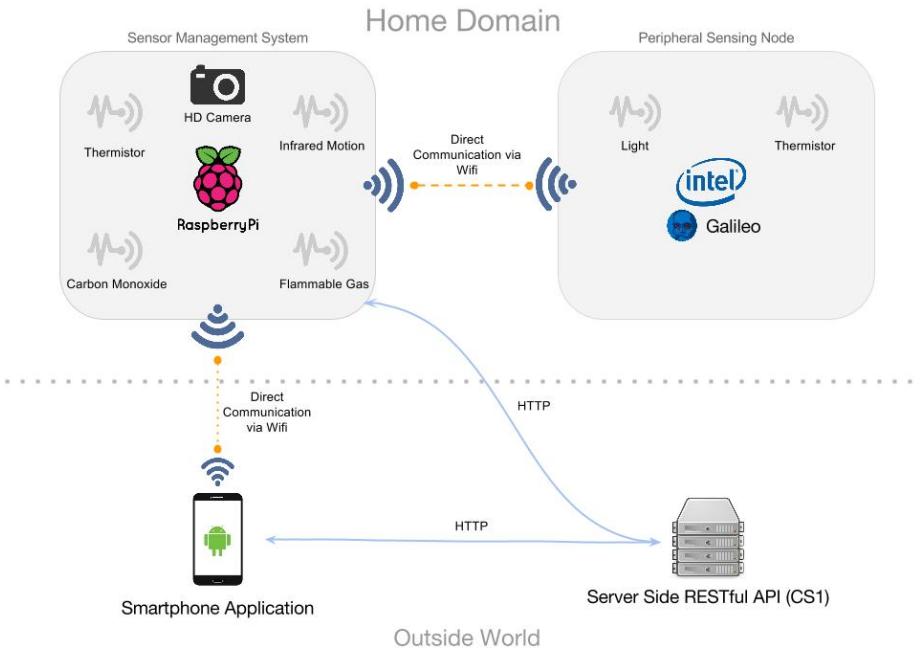

Figure 1: High Level System Architecture Diagram

3.2.2.1 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

3.2.2.2 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.

3.2.2.3 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 backend functionality required to service both the Sensor Management System and the Android smartphone 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

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

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.

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]

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

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 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

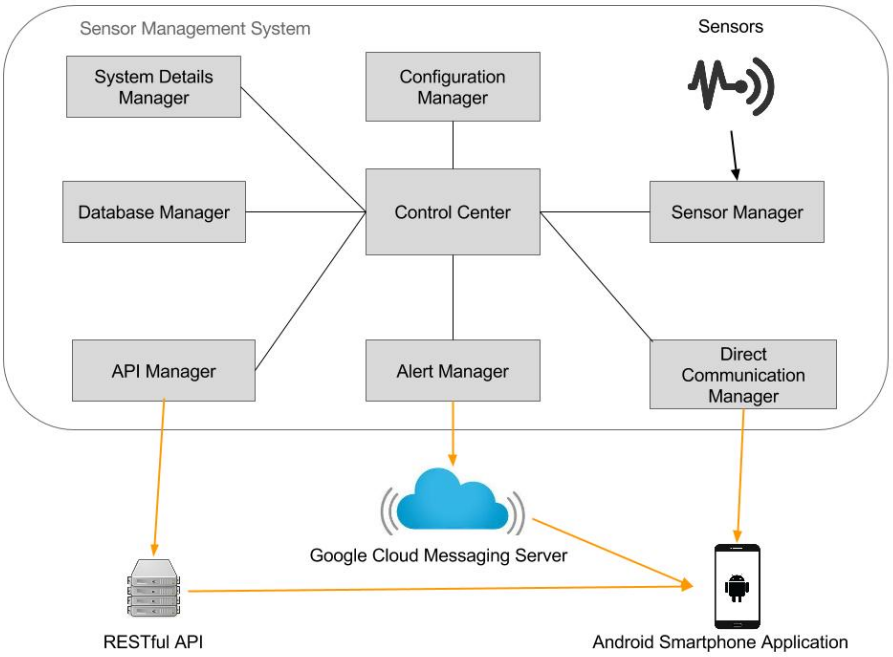

Figure 2: Sensor Management System High Level Class Diagram

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.

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 2 facilitates high level sensor orchestration. Its incharge 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 2 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 4.3.1. 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 then push notification requests, it also handles sounding of the buzzer, and camera operation.

4.3 Android Application

4.3.1 Push Notifications

References