Industrial Applications

Beacube: prototype report

Master's in

Embedded Computing Systems

Academic year: 2015/2016

Submitted by

Pasquale Antonante

Elena Lucherini

University of Pisa, July 3, 2016.

Contents

1 Table of abbreviations and definitions

Beacube: name of the developed system

BLE: Bluetooth Low Energy (Bluetooth 4.0, or Smart)

RSSI: Received Signal Strength Indicator

Trigger: event triggered by the system

Trigger zone: range of action of the Bluetooth central

2 Introduction

2.1 Smart home

Smart home products are becoming more and more popular, especially in North America, where analysts predicted $200 billion would be spent by young people in the near future. While the focus so far has been mainly on security systems, the smart home is pervading many other fields, such as lighting. Customers are now asking their products for better and easier control, and for help in anticipating their needs.

Our proposal is a smart home system that can provide an automatic behavior and personalized actions.

2.2 The enabling technology: iBeacon

In order to understand our technology, it is first necessary to introduce the main enabling technology that we are going to use: iBeacon [1].

iBeacon is Apple's implementation of Bluetooth low-energy (BLE) aimed at creating an alternative to GPS, capable of providing location-based information and services to a Bluetooth-enabled device.

The beacons themselves are small, cheap BLE transmitters. A Bluetooth-enabled device listens out for the signal transmitted by beacons and responds accordingly when the beacon comes into range.
For example, on a visit to a museum, the app of the museum, installed on a Bluetooth-enabled device (i.e. your smartphone or tablet), would provide information about the closest display, using your distance from beacons placed near exhibits to work out your position. iBeacon is a much better option for in-door mapping – which GPS struggles with.

2.3 Beacube

We want to implement iBeacon into a smart home system in a way that prevents the user from needing their smartphone: entering or leaving the room will trigger actions, such as turning on or off the lights of a smart lighting system; this eliminates the overhead introduced by the use of the proprietary applications of the existing smart lighting systems. The resulting system is called **Beacube**.

Beacube stores each user's preferences and actions and will trigger them whenever the user enters or leaves the range of action. The range, called **trigger zone**, is customizable through a web interface or iPhone application.

Beacube is a cross-platform, open-source product that allows the highest degree of customization. The system provides a mechanism which lets developers and companies create third-party applications and upgrade Beacube's functionalities beyond the ones it was conceived for.

2.3.1 How it works

Each user has their own beacon (a small device, such as a key chain or a wearable), which allows the system to track their habits. The central part of the system is a cube which will be positioned in a room of the user's home. Beacube monitors the surrounding space in search for a beacon (i.e. a user) to trigger the actions chosen by the user as they enter or leave the range of action (i.e. the room).

2.3.2 Prototype

To build a fully-functioning prototype, the following components were used:

- The Estimote iOS application to simulate the beacons.
- A Raspberry Pi 2 with a wireless USB network adapter and a USB Bluetooth 4.0 adapter.
- Electronic components, such as wires, photoresistors, trimmers and LEDs to simulate a smart home lighting system to control.

The development of the prototype was broken down into the following main steps, with the implementation of:

1. a module to scan and discover nearby Bluetooth Low Energy peripheral devices. The Node.js Bleacon module was used. A filter was designed to cancel the noise from the RSSI (Received Signal Strength Indicator) and compute a more accurate estimate of the distance of the peripherals.
2. the infrastructure of the service in Node.js, including a RESTful server to make all the functionalities available to the clients, a process and log management, and the integration of multi-cast DNS service discovery.
3. a database in order to keep track of the registered users, their actions and when they get triggered.
4. a mechanism to allow custom actions built from third-parties.
5. web and iOS applications to register, change the settings and upload custom actions.
6. an electrical circuit and the Node.js modules required to control it to build a case of application: the goal is to simulate a smart home lighting system.

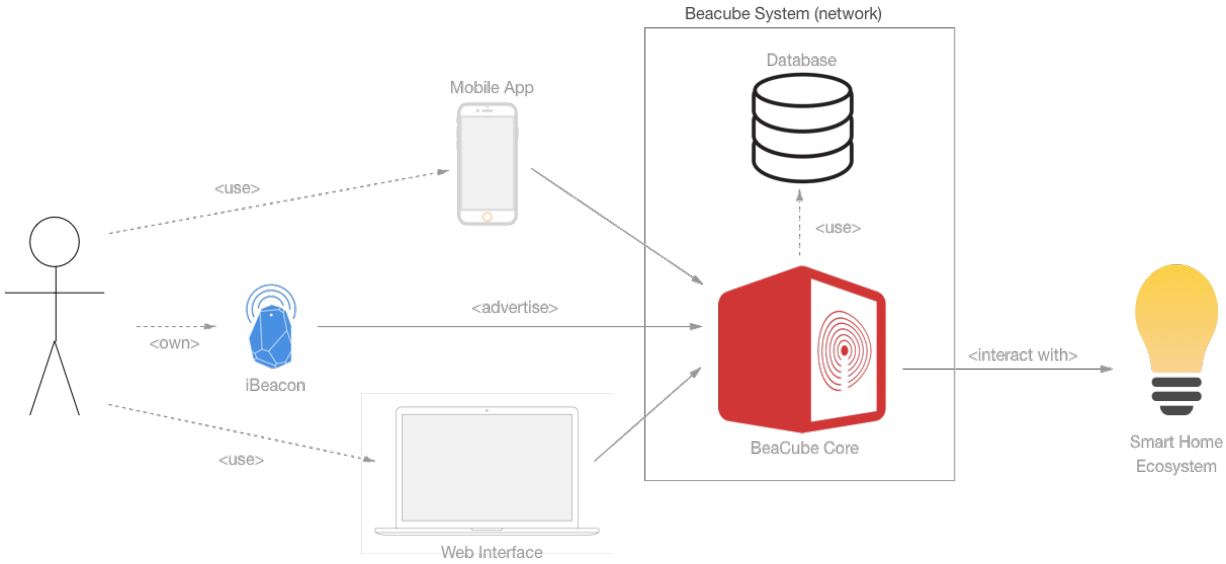


Figure 2.1: Diagram of the system structure

2.3.3 System overview

The user can interact with the system by means of two interfaces: a web application or an iOS application. Through the interfaces, the user can register their peripheral device (beacon) and assign a user name to it, and subscribe to (or unsubscribe from) the actions to take. It is also possible to define the distance range (trigger zone) within which the actions can be triggered. The recommended trigger zone is such that the actions be triggered as the user walks in or out the room in which Beacube is located.

The goal of the prototype built for the demonstration is to manage a system with two users and turn on the red or green led depending on the user's preferences. The user can also adjust the light's brightness to their taste.

Beacube allows third-parties to develop and upload new actions to be triggered as the user walks in or out of the trigger zone. The new actions must be written according to the requirements specified in Section 6.1, and can be uploaded from the web application, in the modality described in Sections 7 and 6.

3 Scan and discovery

3.1 Bleacon

Bleacon [2] is a Node.js library for creating, discovering, and configuring iBeacons, based on Noble (a module that allows the developer to build a Bluetooth Low Energy central).

In this implementation, as soon as the central on which Bleacon is running is powered on, it starts scanning the surrounding space in search of beacons.
When a beacon is discovered, it is stored into an array, `userBLE`, which keeps track of the connected beacons. In particular, `userBLE` is an array of `UserBeacon` objects, which store, for each beacon:

- the UUID (Universally Unique IDentifier),
- the RSSI (Received Signal Strength Indicator) detected,
- the time of the last detection,
- the user to which the beacon is associated. If the beacon is not registered yet, the field is set to `null`. This field is an instance of the `User` class, described in Section 4.2.1.
- the computed distance. The method is described in details in Section 3.2.

When a beacon is discovered that has already been stored in the array, Bleacon updates its distance. The beacons that have not been detected by the Bleacon module for more than three minutes are deleted from `userBLE`.

3.2 Distance computation

Estimating the distance of a beacon can be difficult [3]: for a beacon that is 5 meters away, distance estimates might vary between 2 meters and 10 meters. The biggest factor affecting error in distance estimates is radio noise.

The transmitter power field of a beacon indicates how strong the signal should be at a known distance, through the RSSI. Values of RSSI are measured in dBm, and a typical one-meter calibration value for a beacon transmitter is -59 dBm.

Greater negative numbers of RSSI represent weaker signals, while smaller negative numbers represent stronger signals: if the RSSI of a beacon is greater than our calibration value, this means the beacon is probably more than one meter away.

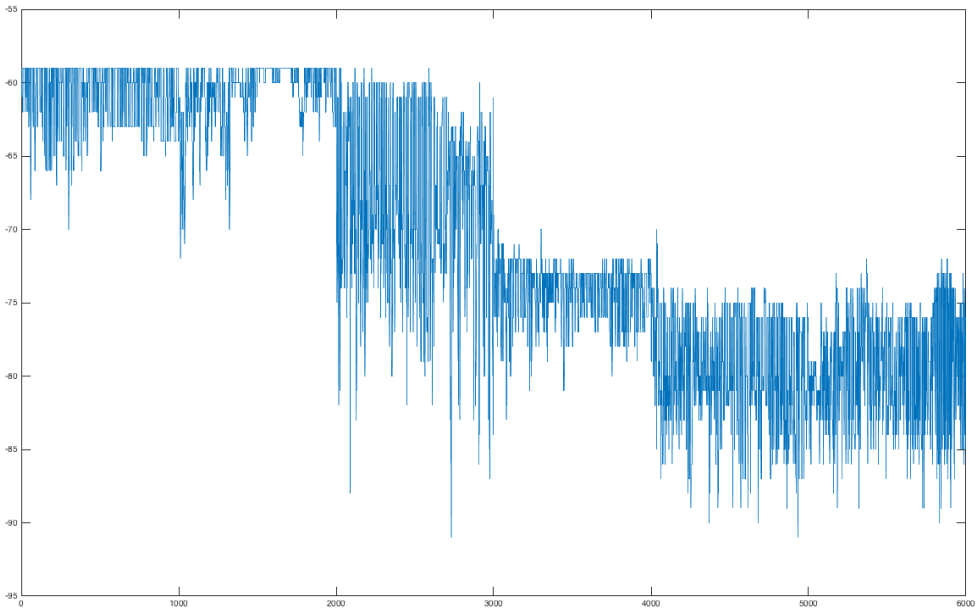


Figure 3.1: Raw data: RSSI measured at 1, 2, 3, 4, 5, 8 an 10 meters.

In Beacube, the raw data were obtained by measuring the RSSI at distances between 1 and 10 meters (namely, at 1, 2, 3, 4, 5, 8 and 10 meters).

3.2.1 Noise filtering

The simplest way to filter out this noise is to use a running average of RSSI measurements — e.g. the most recent 20 seconds worth — to help smooth out the noise. It is also common to discard particularly high or particularly low numbers in these data sets before averaging.

The disadvantage of filters is the delay they introduce to get the distance estimates. The method described would converge in 20 seconds - an unacceptable delay in this application. For this reason, particular attention was dedicated to the noise filtering. Four filters were designed and compared, using four different techniques: a lowpass filter, a Kalman filter, a Savitzky-Golay filter and a moving average filter [6].

Kalman filter

The Kalman filter [4] is a state estimator that makes an estimate of some unobserved variable based on noisy measurements. It is a recursive algorithm as it takes the history of measurements into account. The regular Kalman filter assumes linear models. The resulting system is as follows:

$$x_k = A \cdot x_{k-1} + Q$$
$$z_k = C \cdot x_k + R$$

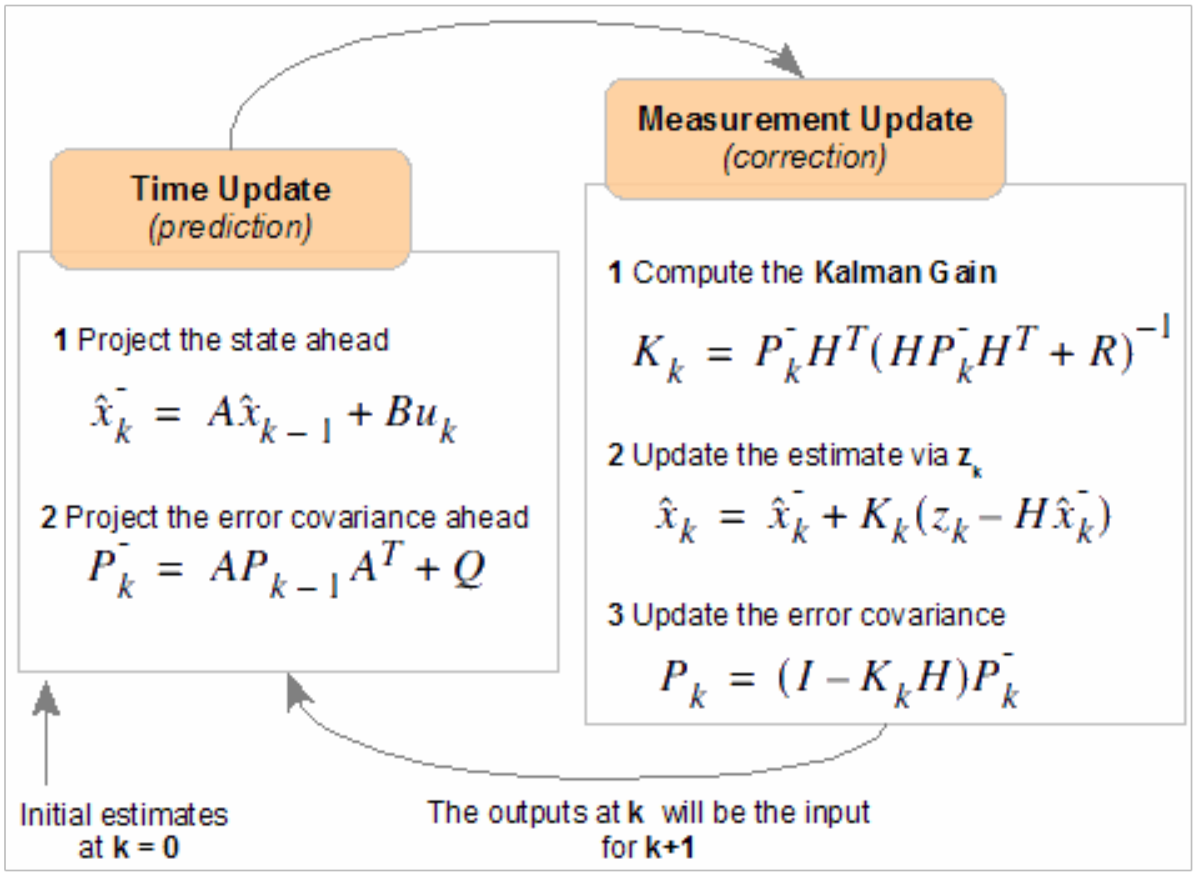


Figure 3.2: Kalman filtering process

The current state x_t is defined as a combination of the previous state x_{t-1}, given some transformation matrix A, and the process noise Q (that is, the noise caused by the system itself). In our measurement, we assume that the device does not move, which means that over time, we expect a constant RSSI signal. To reflect this, A must be set to the identical matrix.

The second equation represents the observation model: C is the transformation matrix and R is the measurement noise, caused by inexact measurements. Since we model RSSI directly, C must be set to the identical matrix.

Figure 3.2 shows the filtering process. R can be computed as follows: keeping the sensor in a static position, we measure the values read by the sensor. We then compute the covariance on those values. Q can instead be found by trial and error.

Savitzky-Golay

Savitzky-Golay [5] is a method of data smoothing based on local least-squares polynomial approximation. The basic idea behind it is that fitting a polynomial to a set of input samples and then evaluating the resulting polynomial at a single point within the approximation interval is equivalent to discrete convolution with a fixed impulse response.

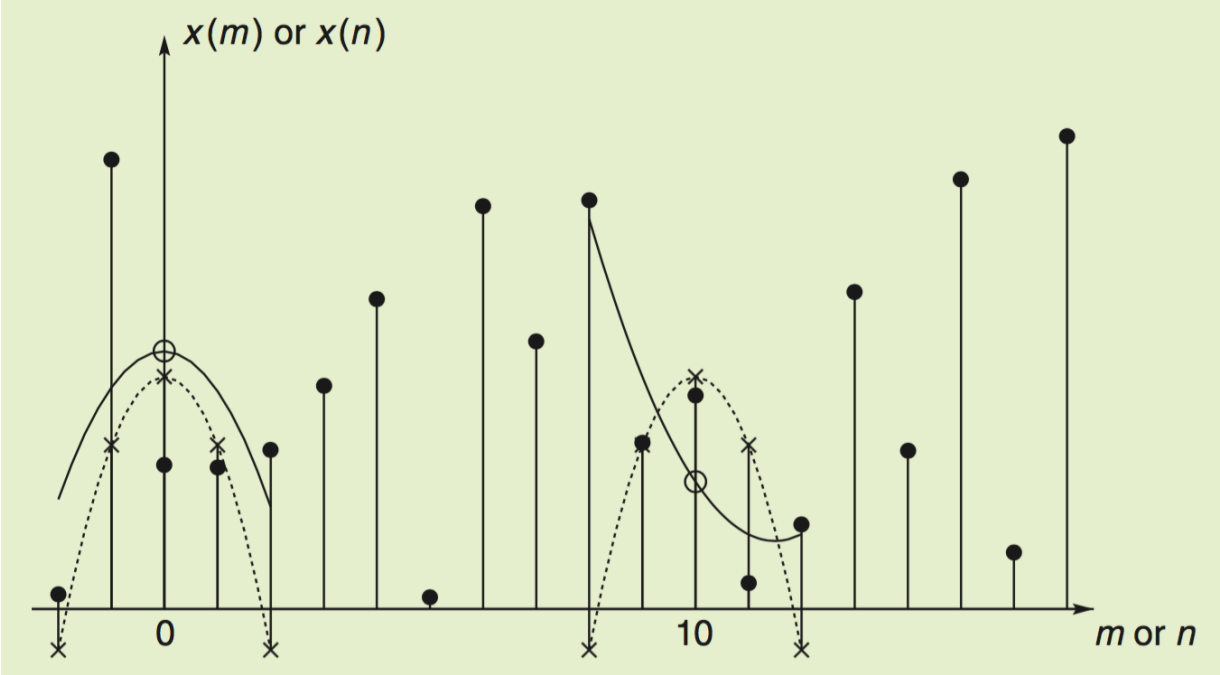


Figure 3.3: Least squares smoothing by locally fitting a second-degree polynomial to five input samples.

To apply Savitzky-Golay, it is necessary to choose the window size and the degree of the polynomial interpolating the collected data. A process is then executed to compute the coefficients of polynomial, using the dataset, along with its values at the data points.

Figure 3.3 shows the idea behind Savitzky-Golay, fitting a second-degree polynomial (solid lines) to five input samples. The black circles denote the input samples, the empty circles denote the least-squares output sample, and the crosses denote the effective impulse response samples (weighting constants). The dotted line denotes the polynomial approximation to centered unit impulse.

The higher the degree of the polynomial, the noisiest the approximation; on the other hand, a low degree results in a smoother signal. It is necessary to find the right trade-off between the two options. In this implementation, the noisy signal was approximated to a second-degree polynomial, as once it goes beyond one of the chosen thresholds (see Section 3.2.2), it does not cross it back.

3.2.2 Distance estimates

The best methods to compute the distance estimates are the method used in the Android Beacon Library and the path loss model proposed by Botta and Simek. The former is obtained with the following equation:

$$dist_{est} = A \cdot ratio^B + C$$

,

where

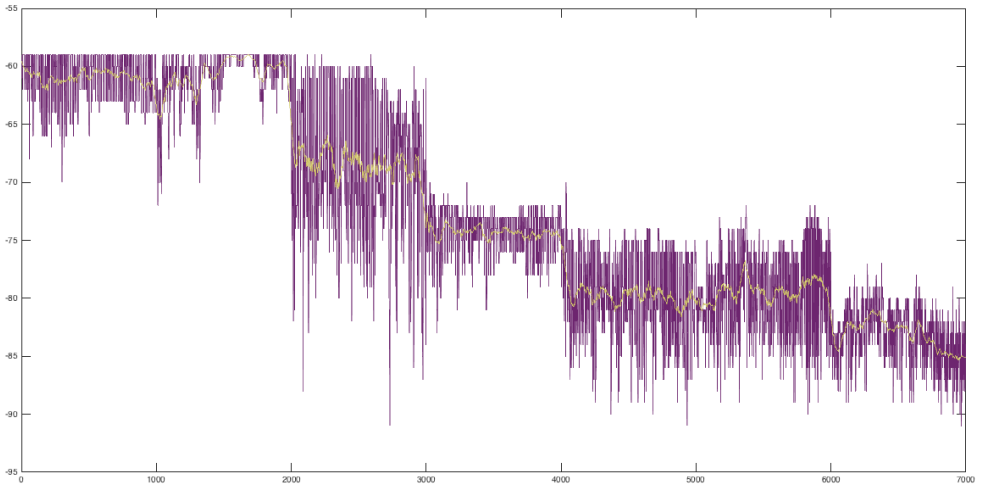


Figure 3.4: In gold: filtered data using the Savitzky-Golay filter.

$$ratio = \frac{RSSI}{TxPower}$$

.

$TxPower$ refers to the reference RSSI at 1 meter (in this case, -59 dBm). A, B and C are computed by means of a genetic algorithm.

The path loss model [7] is based on the (ideal) assumption that the RSSI is a function of the transmitted power and the distance between two radio devices. The resulting formula is the following:

$$d = d_0 \cdot 10^{\left(\frac{RSSI_{d_0} - RSSI}{10\eta}\right)}$$

Where d_0 is the reference distance (1 meter) and $RSSI_{d_0}$ is the reference RSSI (-59 dBm). η is a parameter found and optimized with a genetic algorithm.

The path loss model provides estimates that are more accurate with greater distances. For this reason, the model was preferred over the Android Beacon Library's method.

Choosing the filter

As seen in Section 3.2.1, the distance estimates were computed on the dataset filtered using four different techniques. Overall, the filters gave similar results, shown in Figure 3.5. However, it is possible to appreciate a few differences by zooming in, as shown in Figure 3.6.

The moving average filter transitions more quickly, but the resulting estimates are significantly noisy. The Savitzky-Golay filter has slightly worse performances with respect to

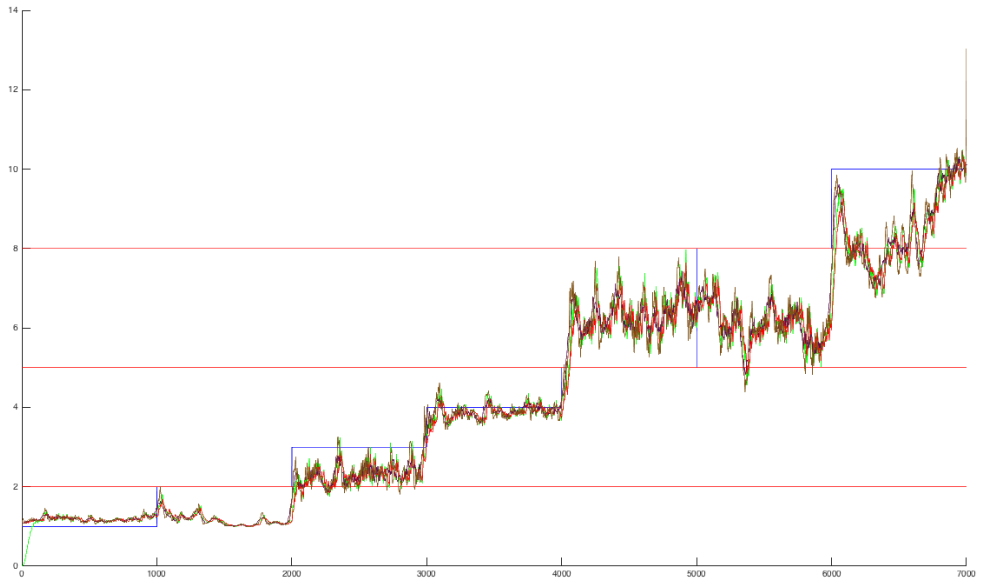


Figure 3.5: Estimates computed with the four filters. The red horizontal lines represent the chosen thresholds.

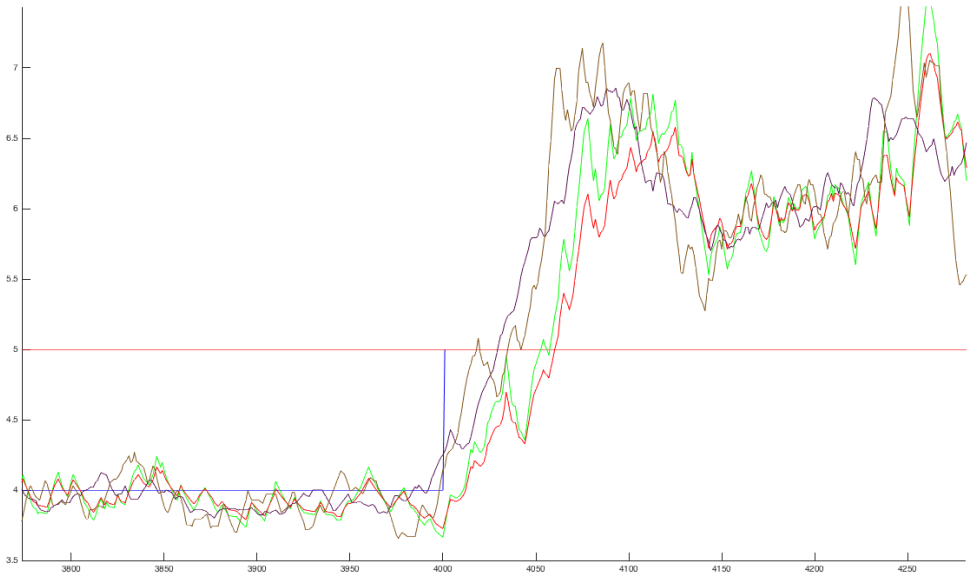


Figure 3.6: Estimates: zoomed in. Brown: moving average filter, green: low pass filter, red: Kalman filter, purple: Savitzky-Golay. The horizontal line represents a threshold. In blue is the reference signal.

speed, but results in more accurate estimates. Both the Kalman and the low-pass filter proved themselves to be too slow for this application.

After comparing the performance of the four filters, the Savitzky-Golay filter was chosen as a trade-off between speed and accuracy.

Thresholds

Please note that, as you can see in Figure 3.5, the distance estimates of 1 meter and 2 meters are very similar. Therefore, the estimate is not accurate enough to distinguish whether the detected beacon is at a 1-meter or 2-meter distance. To reduce the uncertainty, we decided to limit the trigger zones (range of actions that can be chosen within the application) to the Apple iBeacon approximations of *Immediate*, *Near* and *Far*. In our implementation, as shown in 3.5, *Immediate* corresponds to distances below 2 meters, *Near* between 2 and 5 meters, *Far* between 5 and 8 meters. Distances greater than 8 meters are not taken into consideration, as we expect to use Beacube in a home environment. See Section 7 for more details.

4 Infrastructure

The main infrastructure of the system provides mechanisms for the user to:

- associate their beacon to a user name and change their preferred trigger zone,
- subscribe to an action to trigger when the user is within the trigger zone,
- unsubscribe from an action.

In order to do this, a RESTful server is provided. As explained in Chapter 7, the server is consumed by the web and iOS applications to make the service available to the user.

4.1 RESTful server

GET methods. The RESTful server provides the following GET methods. Please note that all the responses are encoded in JSON.

- `/beacons`: lists all the connected peripherals. For each beacon, it returns the UUID, the RSSI, the computed distance, the time of the last detection, the user name the beacon is associated to, and the preferred trigger zone.
- `/beacons/:UUID`: returns the information related to the peripheral specified by `:UUID`.
- `/nearest`: returns the information related to the nearest peripheral.
- `/triggerlist`: lists all the existing triggers. For each trigger, it returns its name.
- `/triggerlist/:UUID`: lists all the existing triggers, specifying, for each trigger, whether beacon `:UUID` is subscribed to it.
- `/group/triggerlist`: lists all the existing triggers, specifying, for each trigger, whether the Group user is subscribed to it (see Section 4.3).
- `/log`: returns the output log file. The log is managed by PM2 (see Section 4.4).
- `/err`: returns the error log file. The log is managed by PM2.
- `/shutdown`: shuts down Beacube.

POST methods. The RESTful server provides the following POST methods:

- `/beacons/:UUID`: takes a JSON object { `"username":` *username*, `"triggerzone":` *triggerzone* } and sets user name and trigger zone of beacon `:UUID` to the desired values. If *username*, or *triggerzone*, is `NULL`, the value remains unchanged.
- `/subscribe/:UUID`: takes a JSON object { `"name":` *name* }. If beacon `:UUID` and trigger *name* exist, it subscribes `:UUID` to *name*. It returns error if `:UUID` is already subscribed to *name*.
- `/unsubscribe/:UUID`: takes a JSON object { `"name":` *name* }. If beacon `:UUID` and trigger *name* exist, it unsubscribes `:UUID` from *name*. It returns error if `:UUID` is not subscribed to *name*.
- `/group/subscribe`: takes a JSON object { `"name":` *name* } and subscribes The Group user to *name*. It returns error if Group is already subscribed to *name* (see Section 4.3).
- `/group/unsubscribe`: takes a JSON object { `"name":` *name* } and unsubscribes the Group user from *name*. It returns error if Group is not subscribed to *name*.

4.2 Triggers

A *trigger* is an action which can be triggered as the user enters or leaves their trigger zone.

To store all the available triggers, an object of the `Trigger` class is instantiated in the main infrastructure of the system. Such class implements an associative array, `list`, containing the functions to be executed when an action is triggered; the keys are the name of each variable that stores a function.

To keep track of which actions the user is subscribed to, an object of the `Trigger` class is instantiated in the `User` class, too.

4.2.1 The User class

This class stores:

- the user name,
- the selected trigger zone,
- the list of actions the user is subscribed to. This field is an instance of the `Trigger` class.

`User` is an extension of the `EventEmitter` class [8], which allows the instances to emit events. This mechanism is used to trigger the actions as the user enter or leaves the trigger zone. When the user enters their trigger zone, the event *in* is emitted. Similarly, when the user leaves their trigger zone, the event *out* is emitted.

When the events are emitted, all the triggers the user is subscribed to are executed with the `apply` method [9]. The name of the event (*in* or *out*) is passed as argument, so the trigger can take different actions depending on the behavior of the user: for instance, a trigger turns on the light when the user enters, and it turns it off when they leave. See Section 6.1 for more details.

4.3 Multi-user management

With Beacube designed to be used in a home with multiple users, a new problem arises: the management of a multi-user environment, where the users have conflicting preferences on their triggers. If more users are detected by the system to be within their trigger zone, they might trigger conflicting events: this cannot happen.

The problem was solved creating a hard-coded, persistent user, *Group*, which takes over whenever the system detects two or more 'active' users. As with regular users, triggers can be assigned to Group by means of the web or iOS applications: the selected triggers represent the settings that every user of the system supposedly agreed to.

4.4 Process and log management

The error management is done by `PM2` [10]. PM2 (Process Manager 2) is a production process manager for Node.js applications which allows developers to keep applications alive forever, to reload them without downtime and to facilitate common system admin tasks.

In this implementation, two scripts were written to have PM2 start and shut down Beacube. Other functionalities implemented through PM2 include the following:

- daemonization of the Beacube application, which runs in the background,
- automatic restart in case of crashes,
- monitoring of the resource usage of Beacube,
- log management in real-time and error logging into a file for successive consultation.
- detailed status of the system with a GET request to `/<ip>:9615`.

4.5 mDNS

mDNS [11] is a Node.js multicast DNS service discovery module. Multicast DNS service discovery is a solution to announce and discover services on the local network.

In Beacube, the RESTful server announces it is running on port 80. As explained in details in Section 7, the user can browse the available servers (Beacubes) on the local network through the iOS application, in order to connect to the desired cube. Here, the user can change the settings of their beacon relative to that cube.

5 Database: NeDB

NeDB [12] is an embedded persistent or in-memory database for Node.js. It uses a subset of MongoDB's API and it is therefore compatible with MongoDB. In Beacube's implementation, the Datastore class is defined and used to keep track of the registered users, their actions and when they get triggered. Such class uses a neDB object to store the information. This way, if it is necessary to replace NeDB with another technology, such as SQL, it is sufficient to modify the Datastore class, without intervening on the main infrastructure.

5.1 The Datastore class

The fields defined in the Datastore class are the following.

- A field `params`: a JSON object representing the options of the neDB instance. `params` is passed to the constructor of Datastore.
- A field `db`: a neDB instance, initialized with the parameters specified in `params`.

The methods of the class are the following.

- `insert`: takes the entry as argument and logs it into the database it is called on.
- `update`: updates the entry chosen by `selector`. `selector` and the new `entry` are passed by arguments to the function. An additional argument, `multiAllowed`, may be used to specify whether there are multiple occurrences for the given selector: in such case, `multiAllowed` must be `true`, otherwise `false`.
- `upsert`: inserts a new entry or updates the entry specified by `selector` if it already exists. `selector` and the new `entry` are passed by arguments to the function.
- `findOne`: takes a query and a callback function as arguments. It executes the query and, if the result is different from `null`, it executes the callback function.

5.2 The format

In Beacube, there are two databases that are saved to the file systems: `usersDB` and `triggersDB`, which log all the activities related to the users and all the triggered actions, respectively.

Each entry in `usersDB` stores the beacon's UUID, the user name, and the trigger zone of choice; each entry in `triggersDB` stores the user name, the state on which the action was triggered (either *in* or *out*), and the time at which it was triggered.

5.3 Mode of operation

Whenever an activity that needs to be logged is executed, an event is emitted on `process`, a global object that can be accessed from anywhere and is an instance of the `EventEmitter` class. On the occurrence of such events, different actions have to be performed.
When a user registers, its entry in `usersDB` is inserted or updated, using the `upsert` method.
When a user subscribes to a new action, its entry in `usersDB` is updated, using the `update` method.
When an action is triggered, a new entry is inserted in `triggersDB`.

6 Customization

As mentioned in Section 2.3, Beacube provides a mechanism that allows third-parties to load and trigger new actions without modifying its core functionalities.

6.1 Prerequisites

In order to use the provided mechanism, the developer has to abide to the following rules:

1. The action must be written in Javascript using Node.js as runtime engine.
2. The name of the file and the name of the function performing the action must coincide. Therefore, only one function is allowed in each file.
3. The function must take an argument. Such argument may be used to distinguish from the actions to be taken when the user subscribes, unsubscribes, enters the trigger zone or leaves it: when the function is called, the value of the argument will be set to `'subscribe'`, `'unsubscribe'`, `'in'`, `'out'`, respectively.
4. It is possible to store additional code other than the function in a folder, if it is required to perform the action.
5. The function must be exported using `module.exports`
6. All the code used in Beacube must be open-source. It is then recommended, although not compulsory, that the code be loaded on GitHub, or a similar service.

The following snippet is an example of an action conforming to the requirements:

```
// file: test.js
var test = function (status) {
   switch(status) {
            case 'subscribe':
                    console.log("SUB");
                    // prints "SUB" when user subscribes
                    break;
            case 'in':
                    console.log("IN");
                    // prints "IN" when user enters trigger zone
                    break;
            case 'out':
                    console.log("OUT");
                    // prints "OUT" when user enters trigger zone
                    break;
            case 'unsubscribe':
```

```
                    console.log("USUB");
                    // prints "USUB" when user unsubscribes
                    break;
    }
};

module.exports = test;
```

More meaningful examples can be found in *custom/Led.js* and `custom/Light.js`. In `Led.js`, the system switches on a led when the user walks into their trigger zone; when the user leaves, the led is switched off. The operations of `Light.js` are similar; the main difference is that the system switches on a led only when the user is within their desired trigger zone and the lighting in the room is dim. See Chapter 8 for more details.

6.2 Directory watcher

All the actions must be stored in the *custom* folder of the system. Beacube provides a mechanism to automatically upload to such folder the code of the action, as explained in Section 6.3.

To notify the system that a new file must be loaded from the *custom* folder, Beacube implements a module that watches for changes on *custom* using *Chokidar* [13]. Chokidar is a wrapper around Node.js `fs.watch` and `fs.events`, which allow the management of file system events. Chokidar solves all the bugs of the Node.js file system management.

When a new file is added to the custom folder, the file is loaded using the `require` function. If the file contains an action according to the requirements in Section 6.1, it will be added to the list of the available triggers.

When a file is deleted, its entries in the *require* cache and in the list of the available triggers are deleted. All possible users who were subscribed to the action defined in the deleted file are automatically unsubscribed.

If a change is detected in an already existing file, its entry in the *require* cache is deleted and the file is included again with an additional call to the `require` function. No changes are made on the list of available triggers, nor on the subscribed users.

Please note that *custom*'s sub-folders are ignored by the watcher.Thus, they may be used to store additional code that might be required by the action to be loaded, as mentioned in Section 6.1.

6.3 Upload

The new triggers can be uploaded through the web interface (see Section 7). Two kinds of formats are supported: Git repositories and .tar archives.

When entering the link to a Git repository in the dedicated page of the web application, the system saves the link and clones the repository in the `/custom` directory. Then, the directory watcher loads the new content in the modalities described in the previous sections.

If a link to a .tar archive is provided, the system downloads the application into `/custom` using the SSH command `wget`, so that the directory watcher load the new content.

Note that all the Node.js dependencies must be included in the Git repository/.tar archive.

7 Web and iOS applications

As mentioned in Section 2.3.3, the user can interact with the system through two interfaces: a web application and an iOS application.

7.1 Web application

The web application consists of five pages:

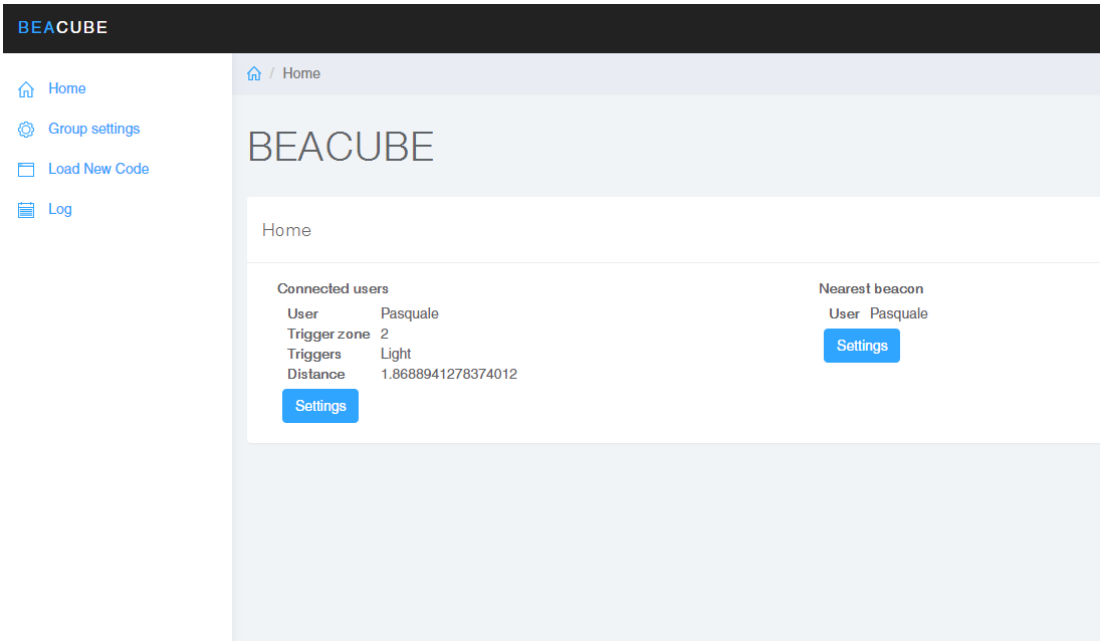


Figure 7.1: Home page

1. **A Home page:** in this page, the list of connected beacons and the nearest beacon are displayed, and updated every 0.5 seconds. The information is obtained through GET requests to `/beacons` and `/nearest`, respectively. If the user has already registered their beacon, the entry shows their user name of choice; otherwise, it shows the UUID of the peripheral. For each connected beacon, a *Settings* button is displayed, redirecting to the Settings page of each device.

2. **A Settings page:** in this page, the user can change the settings of their device. Namely, they can change the user name the beacon is registered to, choose a trigger zone through the slider, and subscribe to (unsubscribe from) the available triggers. The subscription (unsubscription) is saved whenever the corresponding check-box is ticked, through a POST request to `/subscribe/:UUID` (`/unsubscribe/:UUID`). The other information is saved through a POST request to `/beacons/:UUID` whenever the user clicks on the *Change Settings* button.

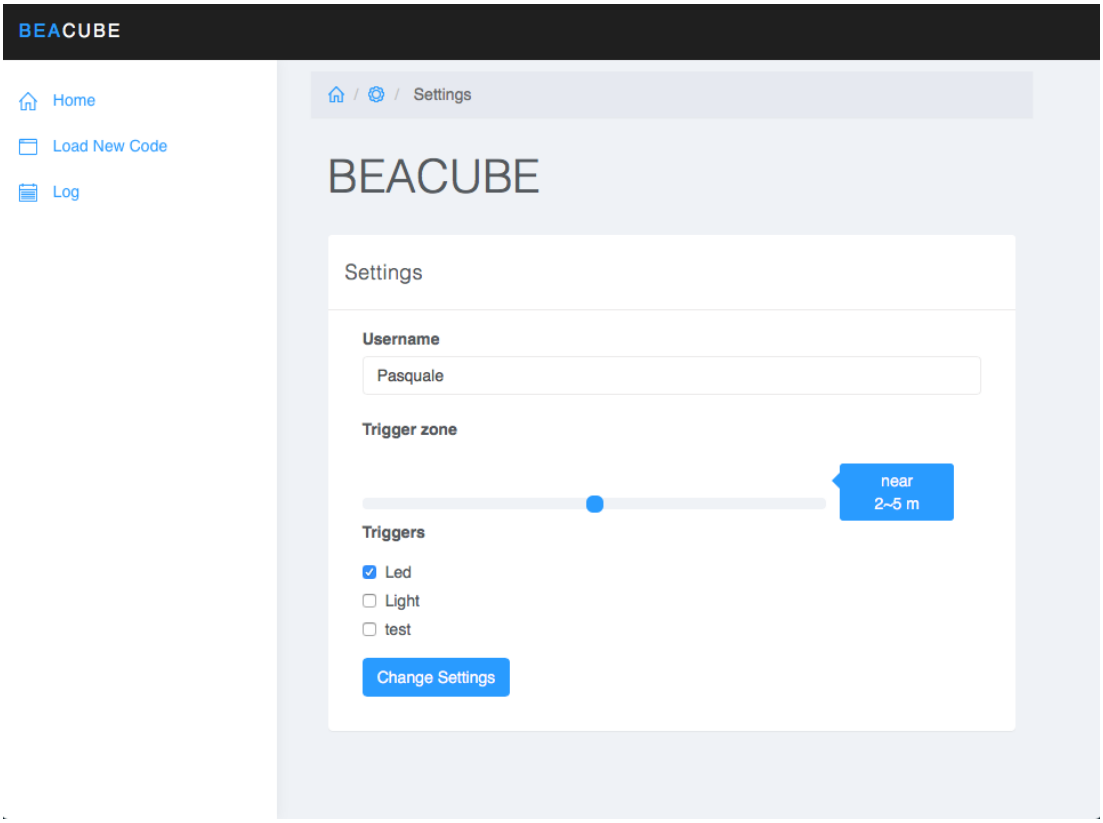


Figure 7.2: Settings

3. **A Group Settings page:** in this page, the user can assign triggers to the Group user. As explained in Section 4.3, the Group user takes over whenever two or more users are detected by the system in their trigger zone. The selected triggers represent the settings that every user of the system supposedly agreed to. The subscription (unsubscription) is saved whenever the corresponding check-box is ticked, through a POST request to `/group/subscribe` (`/group/unsubscribe`).

4. **A Load New Code page:** in this page, the user can avail themselves of the mechanism to automatically upload new code into Beacube and define new triggers (see Section 6.3). The URL of the repository/.tar archive containing the code must be copied into the text box. The corresponding radio button must be selected, specifying whether the link is to a Git repository or a .tar archive. If the check box is ticked, all the connected users will be automatically subscribed to the trigger.

5. **A Log page**: the page displays the output and error logs, retrieved using the GET method on `/log` and `/err`.

7.2 iOS application

When the user opens the iOS application, they are met with a list of all available Beacube centrals. By tapping on the desired one, the user can connect to it, in order to view or change the settings relative to their beacon. Sliding on the entry of a cube, it is possible to turn it off by subsequently tapping on *Shutdown*. The shutdown is managed by PM2, as seen in Section 4.4.

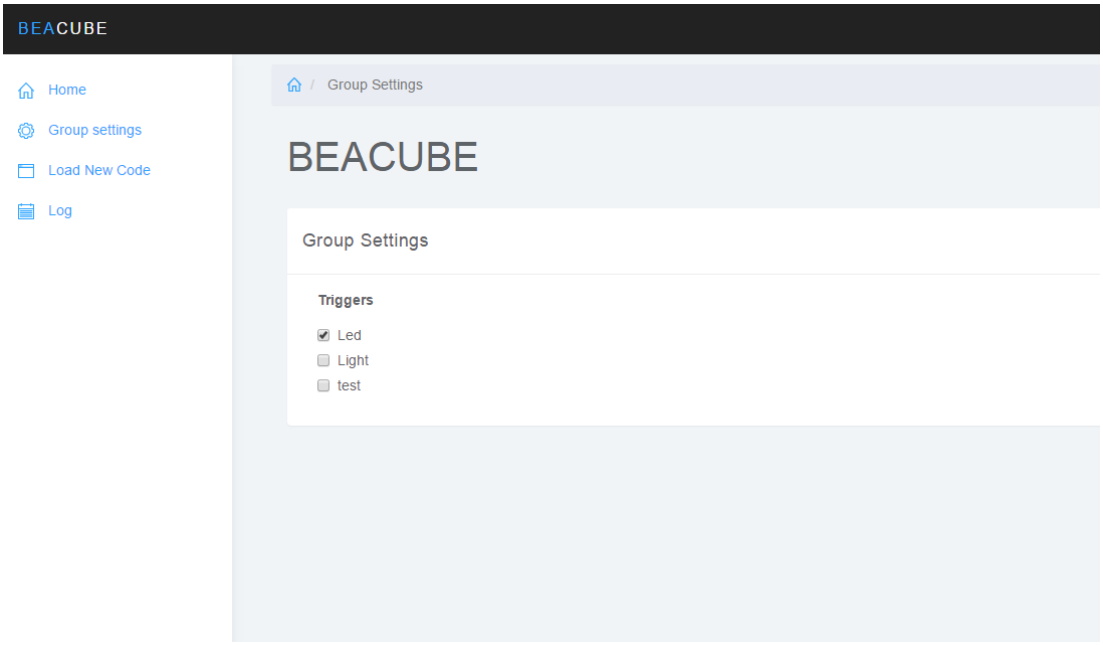


Figure 7.3: Group Settings

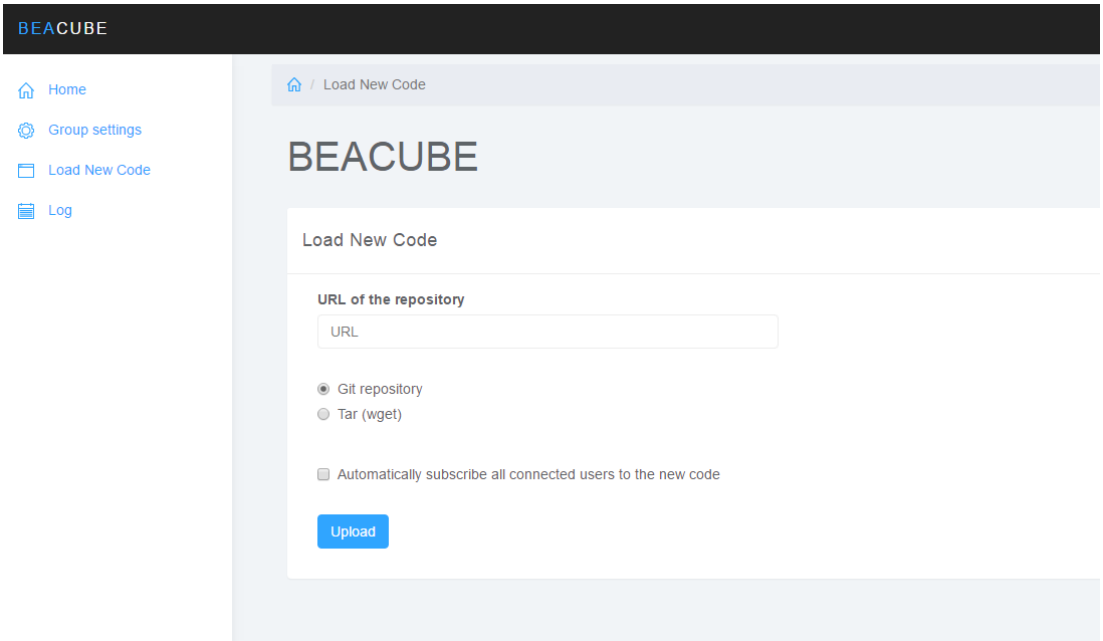


Figure 7.4: Load New Code

In the next page, the user will find the nearest beacon, supposedly the one they want to view or change settings to, by tapping on the green circle. This is done through a GET request to `/nearest`.

When the system has retrieved the nearest beacon, it returns its settings (GET request to `/beacons/:UUID`). The settings may be changed from this screen, according to the mode of operations described for the web application.

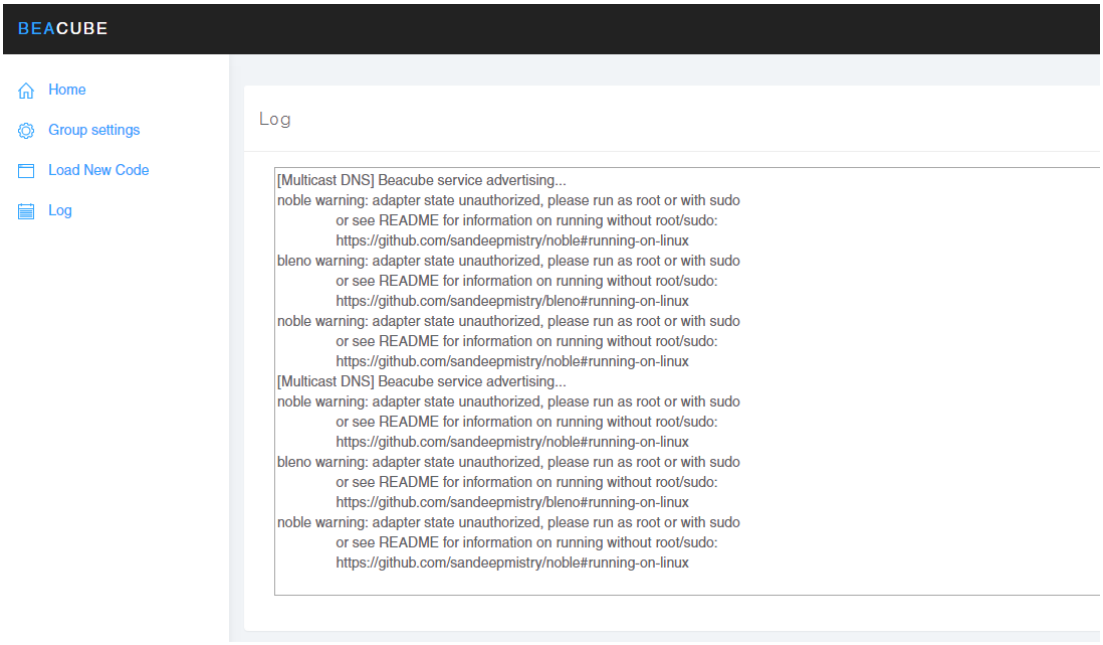
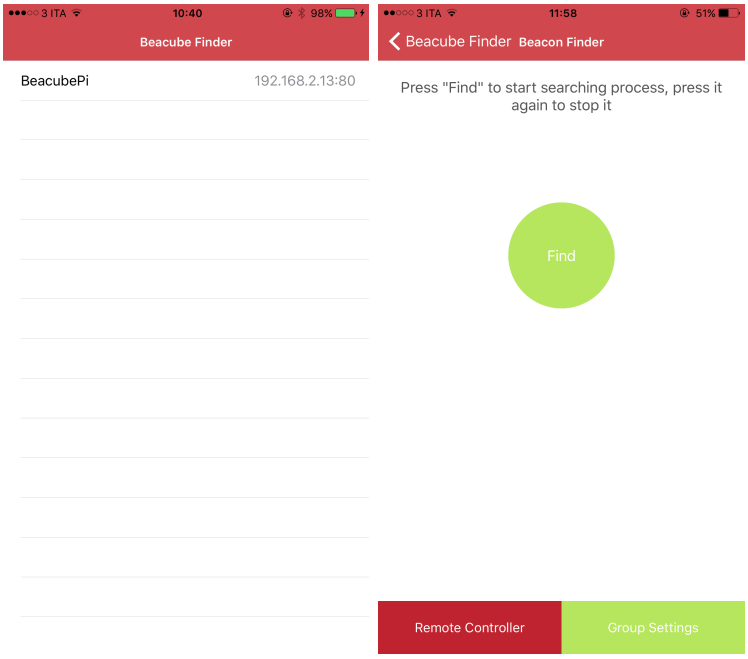
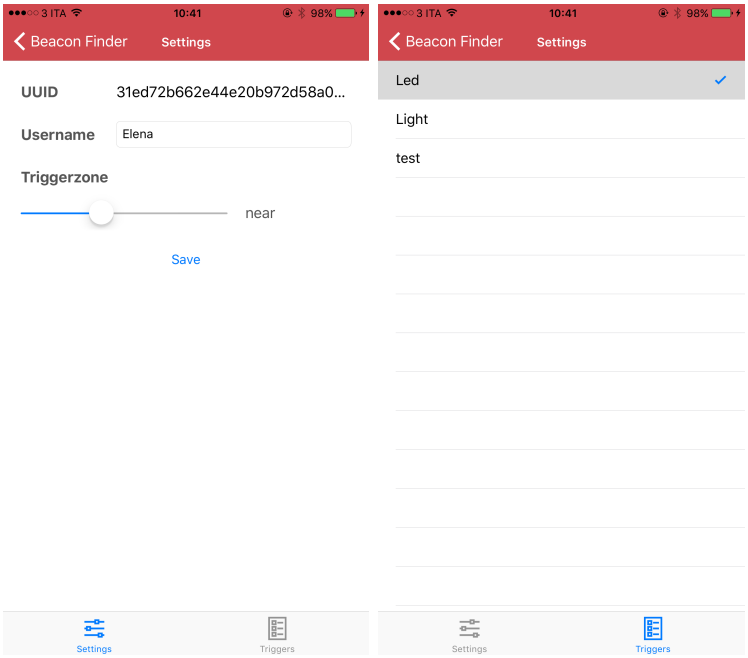


Figure 7.5: Log page



From the Beacon Finder page, it is also possible to access the remote control (see Section 8) and the Group settings. The latter is similar to the Group Settings page of the web application.

It is not possible to use the iOS application to upload new code to Beacube.



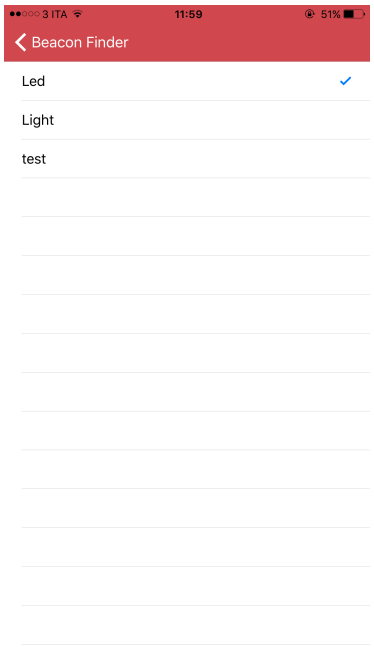
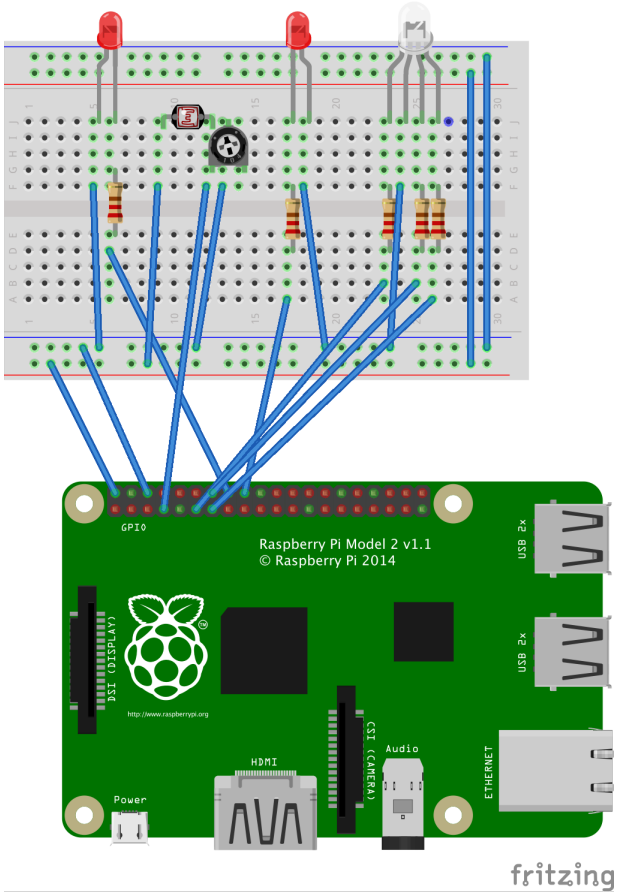


Figure 7.8: Group settings

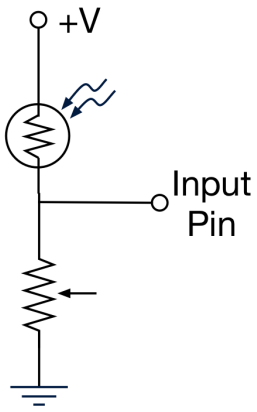
8 Demonstration



The components mentioned in Section 2.3.1 were used to build a case of application: a circuit to simulate a smart home lighting systems. Two triggers are required to enable it: `Led` and `Light`. If the user subscribes to the `Led` trigger, the connected led is switched on as they enter the trigger zone. The led is turned off as the user leaves the trigger zone.

If the user subscribes to `Light`, the connected led is on when the user is in the trigger zone and the brightness detected by the photoresistor is above a given threshold. Conversely, the led is off when the user is out of the trigger zone, or the brightness detected is below the threshold.

The `Light` trigger requires the use of an analog-to-digital converter. The circuit was designed using a Raspberry Pi 2, which does not include a hardware ADC.



The figure above shows how to use a trimmer for a fast fix to the ADC problem (an alternate solution is to use an actual, external, converter). The resistance in the photoresistor, connected to ground, changes with the brightness level. The trimmer is installed and manually set to a resistance value such that the digital value read from the input pin is "0" when the detected brightness is high, "1" when it is low. In `Light`, the led is switched on only if the value read from the pin is equal to "1" (granted that the user is within the trigger zone).

Finally, to enrich the demonstration with an additional feature, a remote controller interface was added to the iOS application. The remote for the RGB Led application is shown in Figure **??**. The user can choose a color through a color picker. The color is then assigned to an RGB led: this is possible thanks to an additional POST call, named `rgb`.

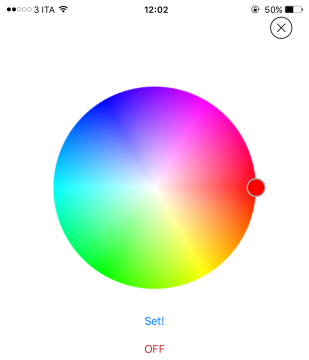


Figure 8.1: Remote control of RGB Led

`rgb`, `Led` and `Light` make use of the *onoff* Node.js module, which allows the developer to control the Raspberry Pi GPIO pins. Additionally, `rgb` uses the Node.js `pi-blaster` library for an easy control of PWM.

9 Conclusions

In this paper, we described a simple prototype developed to showcase the main functionalities of the Beacube system.

Building from the demonstration, we expect the final project to be a distributed system that can control existing smart home products, in order to create a cross-platform ecosystem. A possible solution to bridge the gap is to implement triggers with NetBeast [14]. NetBeast is an Internet of Things Javascript platform that allows developers to create and deploy applications without having to worry about learning different proprietary APIs. This way, a single application can be developed for an entire class of smart products (e.g. smart lighting systems) and work on many different devices that belong to the same category.

To turn Beacube into a distributed system, it is sufficient to modify the Datastore class to advertise the changes to all the cubes (their IP addresses are discovered using mDNS). A possible decentralized database to use instead of neDB is Apache Cassandra [15], a scalable and highly dependable open source NoSQL database.

9.1 Barriers

The beacon technology sets limits that are not easy to overcome. As explained in Section 3.2.1, it is necessary to filter the received signal strength indicator (RSSI) FINIRE for the noise to obtain more accurate and stable distance estimates. The disadvantage of this approach is that, as the user moves relative to the central, it will take a few seconds for the distance estimate to catch up with where it moved.

As beacons get further away from the central, the estimates get progressively less accurate. Thus, it is unrealistic to expect distance estimates to change in near real-time as a user moves around.

Despite these limitations, it is still possible to use the beacon technology to perform the tasks that are at the core of this project: trigger an action to happen when a beacon is very close by, or determine which of several visible beacon is significantly closer (after all, Beacube is built to be used indoor, where distances are within a few meters).

More accurate distance estimates can be obtained with complex data analysis and noise canceling techniques. It is also possible to use triangulation and trilateration for indoor mapping.

9.2 Additional applications

In addition to the consumer application just described, Beacube can be used in logistics and supply chains to track movements of users or items in critical areas, such as warehouses. Currently, companies make use of the NFC technology, which requires the user to touch the reader with a writing device. A network of Beacube systems may provide a more accurate and *natural* tracking as the user/item moves around, without requiring them to approach specific locations. Furthermore, the tracking is enhanced with customized triggers, which let users automatically interact with the items: for instance, if a container is sitting for too long, an operator is automatically alerted.

Bibliography

[1] iBeacon
https://developer.apple.com/ibeacon/

[2] node-bleacon
https://github.com/sandeepmistry/node-bleacon

[3] Radius Networks: Fundamentals of Beacon Ranging
http://developer.radiusnetworks.com/2014/12/04/fundamentals-of-beacon-ranging.html

[4] Estimation and Tracking: Principals, Techniques, and Software
Yaakov Bar-Shalom, X. Rong Li

[5] What Is a Savitzky-Golay Filter?
Ronald W. Schafer, IEEE Signal Processing Magazine, July 2011

[6] The Scientist and Engineer's Guide to Digital Signal Processing
Steven W. Smith, 1997

[7] Adaptive Distance Estimation Based on RSSI in 802.15.4 Network
Miroslav Botta, Milan Simek
http://www.radioeng.cz/fulltexts/2013/13_04_1162_1168.pdf

[8] EventEmitter
https://nodejs.org/api/events.html

[9] MDN - apply()
https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Function/apply

[10] PM2
http://pm2.keymetrics.io/

[11] mDNS
https://github.com/agnat/node_mdns

[12] NeDB
https://github.com/louischatriot/nedb

[13] Chokidar
https://github.com/paulmillr/chokidar

[14] NetBeast
netbeast.co

[15] Apache Cassandra
http://cassandra.apache.org/