university of groningen

faculty of mathematics and natural sciences

A Web-Based Interface for Managing Smart Offices

Bachelor thesis

June 2013

Student: M. Woudt

Primary supervisor: Dr. A. Lazovik

Secondary supervisor: Prof. Dr. ir. P. Avgeriou

Daily supervisor: I. Georgievski

Abstract

With smart offices managing our devices we have no control over our devices. How do we manage these devices? We present a prototype of a web based interface for managing smart offices. The prototype succeeded in being an intuitive interface, and can be used as a base for a more complete interface.

Contents

1 Introduction

We get flooded with devices, at home, the office and all other environments. As much as we like all these devices, they also have their downsides. We spend many hours per year only on interacting with these devices, while they are supposed to make our lifes easier. Besides that, with every device added we use more of one of the most scarce resources on this planet: energy.

Smart environments are designed to address both problems, reducing interaction and energy consumption, by controlling devices. The word 'smart' in smart environment refers to the ability of a smart environment to adapt to user preferences. With this it minimizes the time spent on interacting with devices, and helps you in saving energy. But they are not perfect yet, and we still need to be able to manage the devices overriding the system.

On the other hand, we also need to manage the smart environment itself. While plenty of data can be retrieved from sensors, it is likely that fine-tuning of input data remains important. For example when scheduling a task, the system could perhaps not know that there will be a system outage, or perhaps that there is need for a human being nearby when the task is executed.

With more and more devices in a smart environment, there is a strong need for an easy and intuitive way to manage these devices. We focus on building an interface that is specifically designed for smart environments at offices, smart offices. The system will however be easily adapted to any other smart environment. We will try to come up with a prototype of a functional multi platform interface that is intuitive to use. This prototype could serve as a base for a full-fledged interface to any smart environment.

1.1 Structure

This thesis is defined in seven chapters. After this introduction, chapter 2 gives a short introduction to Smart Environments and in particular Smart Offices. In chapter 3 we look at two projects that are going on in the same context. We take a look at the target audience, requirements and architecture in chapter 4. In chapter 5 we discuss the implementation phase. We start off with looking at the tools that have been chosen, and continue with the three main components of the system. Chapter 6 contains the exercises for the usability test, and a discussion on the results of this test. Finally we come to a conclusion in chapter 7, whereas we also provide some insight in what could improve the system in the future.

2 Smart environments

A smart environment is able to acquire and apply knowledge about an environment and also to adapt to its users to improve their experience in that environment[6]. A smart environment receives information about the user through sensors, and reacts with actuators. An example could be a dark room. When sensors detect a user walking into the room, the smart environment turns on the light. When all users have left the room, the smart environment turns off the light. Here the smart environment helps in the repetitive tasks, and also could help in saving energy in case one would forget to turn off the light otherwise. Smart environments refer to many places, for example, homes, offices, roads and public transportation, but we focus on smart offices here.

2.1 Smart office

A smart office is a smart environment specially designed for use in offices. The concept of smart offices is similar to any other smart environment, but with different constraints. A few important aspects of a smart office are[6]:

- Many different users use the same set of resources
- Only used actively within business hours
- Users are hardly concerned with energy usage
- It should increase productivity
- Can be an extraordinary large number of devices

These aspects have to be taken into account when developing any system related to smart offices. When designing an interface we do not really mind about the first two points, but it should have a benefit in the latter three points. Especially the last point is something that needs to be thought of before starting to build an interface to ensure that the stability of the system is guaranteed.

3 Related work

At the time of writing, there is no other web based interface for managing smart offices publically available. There are two projects that are related to smart offices, but they are mainly focused on making the office and other buildings more energy efficient. Both projects could profit from an easy-to-use web based interface for managing a smart office.

3.1 Energy Smart Offices

The ambitious goal of the EnSO project is to couple, for the first time, advanced research and novel techniques in ambient network technology, probabilistic activity and artificial intelligence planning with innovative service-oriented approaches, thus developing a truly energy-aware platform for the offices of tomorrow[10].

While a lot of energy can be saved by choosing the correct materials and structure for a building, it is not the only way to save energy. With many devices around us, at home and office, a lot of energy can be saved when handling the needs of the user more efficiently. The EnSO project focuses on the huge research gap between the systematic way of controlling devices (e.g., a light that turns automatically off after a certain time of no movements), and controlling the behaviour.

3.2 GreenerBuildings

GreenerBuildings aims to realise an integrated solution that addresses the challenge of energy-aware adaptation from basic (energy harvesting) sensors and actuators, up to an embedded software for coordinating thousands of smart objects with the goals of energy saving and user support[5].

The GreenerBuildings project is based on ubiquitous computing, meaning that computers are everywhere and anywhere. Assuming there are many sensors available, the system will detect humans and their needs and react accordingly. All devices that are not used by these people can be disabled and in the end save energy and money.

The architecture of the GreenerBuildings project can be seen in Figure 1. It is important to see that even with these many sensors available, the system is designed to allow the user to control the system.

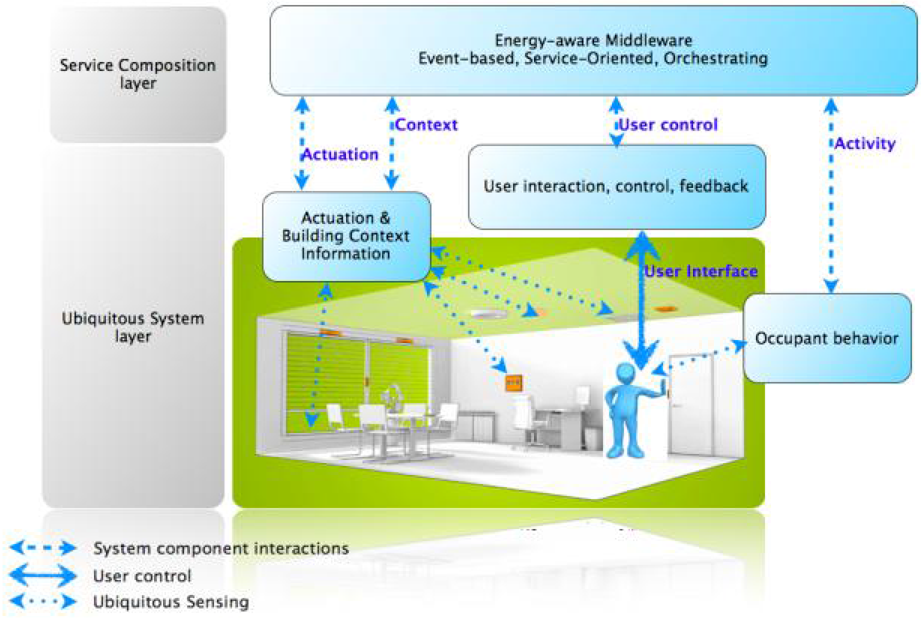

Service Composition layer
Energy-aware Middleware
Event-based, Service-Oriented, Orchestrating
Actuation
Context
User control
Activity
User interaction, control, feedback
Actuation & Building Context Information
User Interface
Occupant behavior
Ubiquitous System layer
System component interactions
User control
Ubiquitous Sensing

Figure 1: GreenerBuildings architecture

4 Design

When designing this interface, we try to design it from a user-centered perspective. With every step we try to imagine where a user would expect a certain functionality. Also we define how the currently existing architecture integrates with the new interface.

4.1 Target audience

The system is designed to be used by the office energy manager or person assigned with the energy managing task. This manager has a good understanding of the devices that are used in the office, so he can make decisions on when to disable devices for saving energy.

While it is only used by a limited number of people, it should be designed with a user centric approach. Most used functions should be quickly available and logically arranged. This supports the productivity in the office and increases the likeliness for the system to be used regularly. Also, in case the office energy manager is not available, any user should be able to perform the required tasks.

4.2 Requirements

The minimal set of requirements is given in the following subsections. It should be noted that additions are possible, but not strictly required, and therefore are not listed.

4.2.1 Functional requirements

1. Device overview

    - There should be an overview of the devices connected. The required information per device is:
        - name
        - location
        - current energy usage
        - maximum energy usage

2. Switch devices on/off

3. Live update of device power usage
4. Display device statistics
    - Display statistics per device per day
    - Display statistics per device per week
    - Display statistics per device per year
5. View schedule
6. Create new schedule item
7. Modify schedule item
8. Delete schedule item

4.2.2 Non-functional requirements

- User friendly
- Multi platform
- Scalable
- Reliable
- RESTful

4.3 Architecture

As given in the non-functional requirements above, the system should be multi platform. Two approaches can be used here, either writing software that compiles for every platform, including some platform specific code, or use a system that is available on almost any system, an HTML website. With the first being expensive in both development and maintenance costs, we have chosen the second approach. Today almost every device has an HTML compatible web browser, and it is likely that any system in the future has a web browser that is compatible with the current HTML standard. Also the maintenance costs are significantly lower than other approaches, because there is no platform specific knowledge required.

A web interface typically consists of two parts: a web server and a web client. The first runs at a server, providing the files and data over a connection, which is usually HTTP or the secured version HTTPS. The latter is the part that runs on the device of the client, inside the web browser. This is what represents the data, and provides access to the functionality. The web client will connect to only the web server, where the web server will connect to the existing system[7]. This system is connected with the sensors and actuators in a Smart Office. It takes care of signalling the actuators based on these sensors, or on a predefined schedule. In figure 2 this existing system is shown as is, where in figure 3 the web server and clients are added, and thus shows the architecture of the systems combined.

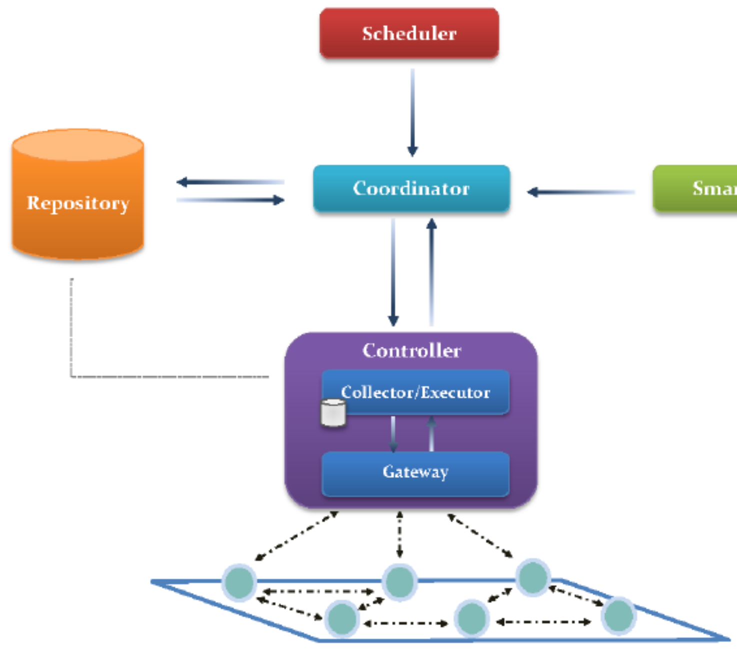


Figure 2: Current system architecture design

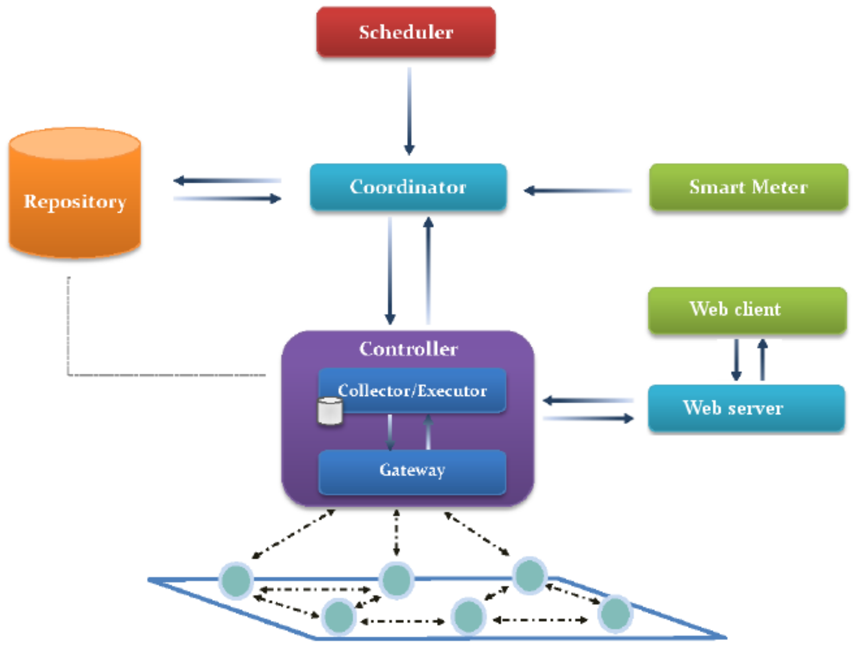


Figure 3: New architecture

5 Implementation

A good implementation starts by choosing the correct tools, therefore these important decisions are discussed first. After that, the three most important elements of the system, the device manager, statistics viewer and the scheduler, will be discussed in depth.

5.1 Tools

5.1.1 Scala

The Scala programming language[3] has proven itself for being highly reliable and scalable. Large companies, including Twitter, Linkedin and Intel, use it for their large mission critical systems[9]. For a smart office, highly reliability is utmost important. Any defect, even the smallest ones, can have a major impact on the continuous workflow of the office. With decreased productivity it will also have a financial impact.

A small office will only have a handful computers, phones, printers, lights and other appliances, whereas a large corporation can have up to millions of appliances. This requires a system that can scale up to almost infinity. The Scala programming language is specifically designed to be scaled for these systems.

5.1.2 Play Framework

Scala is a general programming language, not specifically designed for web applications. The Play Framework[1] is a framework that provides an easy to use architecture for building web applications with Scala. As with Scala, the Play Framework has proven its maturity as it has been accepted by large companies, most notably LinkedIn[4].This framework has a few advantages over other frameworks, including:

- stateless
- lightweight
- asynchronous Input/Output
- written in Scala

The web interface can make good use of the advantages above, and has a stable base when using the Play Framework.

5.1.3 REST

The world wide web is an example of a RESTful system. RESTful is said to be an architecture that is compliant with the constraints defined in the REST style. Representational state transfer[8] (REST) is an architectural style for distributed systems. One of the main goals of REST was ensuring scalability, achieved by being stateless and cacheable.

A RESTful system should have the following constraints:

1. Client-server

    - Client-server architecture is based on the separation of concerns principle. This improves portability of the client side, and scalability of the server side.

2. Stateless

    - Each request from the client to the server should contain all the information needed to process the request. It makes the system more reliable because any system can be replaced without loss of state. Also scalability is improved, because each request can be served to any server available.

3. Cacheable

    - Each request must be labeled implicitly or explicitly as being cacheable or not. Efficiency is greatly improved for static content, however, it has to be used with care to avoid cache data that is different from the server.

4. Uniform interface

    - By providing an uniform interface portability is improved, because same connectors can be used on different interfaces. It also allows independent evolvability.

5. Layered system

- Different layers can exists in a RESTful system, where the client can only see the first layer. With the uniform interface it is easier to implement a new layer. A layer can greatly improve performance, reliability and scalability by using a load balancer.

6. Code-on-demand (optional)

   - This is an optional constraint that does not necessarily has to be supported, but it is allowed. This allows to extend the functionality with scripts that are downloaded and executed at the client. It improves the extensibility of the system, but it does restrict in portability.

5.1.4 Publish-Subscribe pattern

In a traditional web application new data will be fetched every few seconds to keep it up to date. This works perfectly fine for small scaled applications, but would create major problems when many applications try to fetch new data every few seconds. The solution to this problem is the Publish-Subscribe pattern, where instead of requesting new data every few seconds, new data will be pushed to all subscribers when it is available. This produces less overhead and ensures data is pushed to all subscribers as fast as possible, without having to wait for the next fetch cycle.

We can easily compare this to a real life example. Assume you are a news reporter writing for several newspapers. In the traditional scenario someone from the newspaper company would drop by your desk or call you every hour to check if you have a new article. When writing for only one newspaper it would be doable, when writing for many newspapers it is not. With the Publish-Subscribe pattern, you will post your article to a distributor once you finish it, and he distributes it to all newspapers that are interested in articles that you wrote.

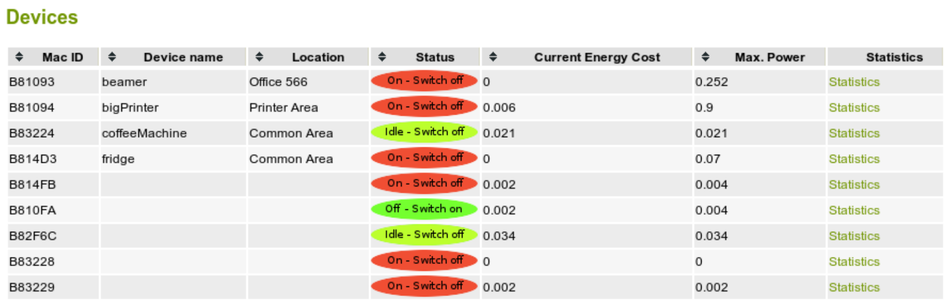
A redis[2] server has been used to implement this behavior, although the collector did not support the push actions yet. To overcome that problem, we have written a redispush application. This application fetches data every few seconds from the source, and pushes it to the redis server. The implementation of this program can be found in Appendix A.

5.2 Device manager

The device manager shows the devices where the user can interact with. It shows the current status of all devices, which includes their MAC id, name, location, state(on, idle, off), current energy usage and the maximum energy usage.The list of devices can be ordered by any of these columns to provide a better overview of the current status. From this list a user can also switch states, switch it off when a device is on or idle, or the other way around. At last the user can directly jump to the statistics view.

The current energy usage is updated as soon as new information about energy usage is provided by the redis server. The redis server pushes the energy usage to the web server, which is subscribed on that channel. The web server and clients implement their own Publish-Subscribe algorithm. The clients subscribe and listen to the web server with the use of HTML5 WebSockets, which is the only valid way to allow two-way communication between the web server and client.

The current interface is shown in figure 3. For this design a minimalistic approach has been used, and shows in a visual manner only the relevant data.



Devices

Mac ID	Device name	Location	Status	Current Energy Cost	Max. Power	Statistics
B81093	beamer	Office 566	On - Switch off	0	0.252	Statistics
B81094	bigPrinter	Printer Area	On - Switch off	0.006	0.9	Statistics
B83224	coffeeMachine	Common Area	Idle - Switch off	0.021	0.021	Statistics
B814D3	fridge	Common Area	On - Switch off	0	0.07	Statistics
B814FB			On - Switch off	0.002	0.004	Statistics
B810FA			Off - Switch on	0.002	0.004	Statistics
B82F6C			Idle - Switch off	0.034	0.034	Statistics
B83228			On - Switch off	0	0	Statistics
B83229			On - Switch off	0.002	0.002	Statistics

Figure 3: Device overview interface

5.3 Statistics viewer

The statistics viewer shows, as the name suggests, statistics about each device. For each device we can get statistics per day, week or year. The statistics are then nicely displayed with the use of jQuery flot, which can be seen in figure 4. While this gives a decent overview of the power usage, it would have been more useful if there was information about actual costs. The system would have to be connected with a smart grid to provide information about energy prices.

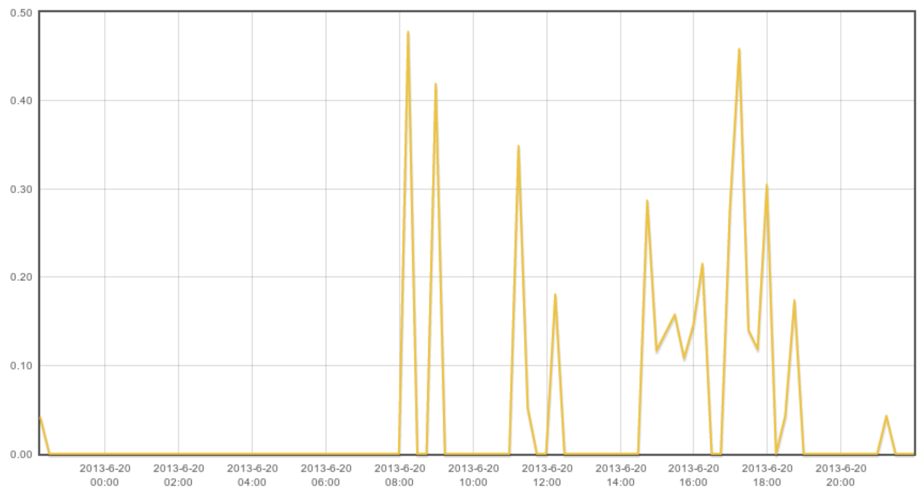


Figure 4: Statistics viewer interface

5.4 Scheduler

The scheduler provides a visual interface to the complete schedule. When the smart office scheduled a certain task, it might not always be scheduled correctly, based on certain context where the system does not know about. In that case it is useful to be able to modify the schedule and fix it accordingly.

Figure 5 shows the interface of the schedule, but also the window that appears when adding a new schedule item. This window is brought up by clicking anywhere on the time scale of a certain device. In case of error, the date and time can be changed. Any change made in this system will directly be visible in the schedule that is still visible in the background, but the schedule item will only be added when the confirm button is pressed afterwards.

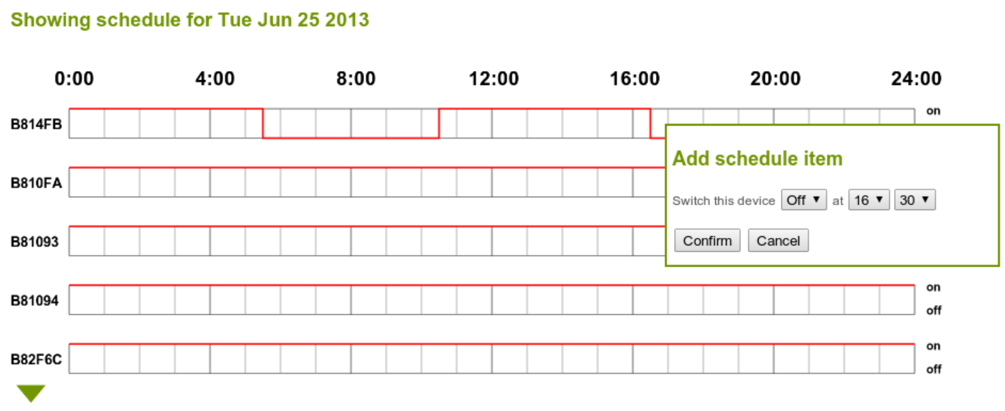


Figure 5: Scheduler interface

6 Usability test

To measure the effect of the user interface, a usability test has been conducted. The tests consisted of three simple tasks, where for each task the time taken, number of steps, and the number of wrong paths taken are noted. There were a total of 14 participants. 7 participants have been given a short introduction, whereas the other 7 have never seen the system before.

6.1 Exercises

1. Switch device with name 'Test Device' off
2. View the daily statistics for device 'Test Device' on June 20th 2013.
3. Schedule device 'Test Device' to switch on tomorrow at 9 PM.

6.2 Results

6.2.1 Exercise 1

The results of the first exercise can be seen in table 1. With the exception of person 8, both groups scored more or less equally. The time and errors columns clearly show that this exercise is easy to accomplish for anyone.

Person	With introduction			Person	Without introduction		
	Time	Steps	Errors		Time	Steps	Errors
1	0:04	2	0	8	0:35	6	2
2	0:08	2	0	9	0:06	2	0
3	0:07	2	0	10	0:06	2	0
4	0:04	2	0	11	0:07	2	0
5	0:12	4	1	12	0:09	2	0
6	0:04	2	0	13	0:05	2	0
7	0:07	2	0	14	0:05	2	0
Average	0:07	2	0	Average	0:10	3	0

Table 1: Result of first exercise of usability test

6.2.2 Exercise 2

1. The results of the second exercise can be seen in table 2. While this test took a bit longer for the participants, the results are not significantly

different between both groups.

Person	With introduction			Person	Without introduction		
	Time	Steps	Errors		Time	Steps	Errors
1	0:11	6	0	8	0:42	12	2
2	0:17	9	0	9	0:15	8	0
3	0:18	10	1	10	0:19	9	1
4	0:14	7	0	11	0:21	8	0
5	0:16	6	0	12	0:12	8	0
6	0:15	8	0	13	0:09	6	0
7	0:11	8	0	14	0:16	7	0
Average	0:15	8	0	Average	0:19	8	0

Table 2: Result of second exercise of usability test

6.2.3 Exercise 3

1. The results of the last exercise can be seen in table 3. While the time to complete the task is almost equal in both groups, we do see an increase in steps and errors for the participants without introduction. While surveying, it is noticeable that the participants in the second group were looking for any guidelines, but with trial-and-error still figured it out pretty quickly.

Person	With introduction			Person	Without introduction		
	Time	Steps	Errors		Time	Steps	Errors
1	0:05	6	0	8	0:22	16	5
2	0:06	8	1	9	0:08	8	1
3	0:08	8	1	10	0:06	9	2
4	0:07	6	0	11	0:07	9	2
5	0:07	5	0	12	0:06	8	1
6	0:08	6	0	13	0:05	5	0
7	0:07	5	0	14	0:06	6	0
Average	0:07	6	0	Average	0:09	9	2

Table 3: Result of third exercise of usability test

7 Conclusion and Future Work

The current prototype succeeded in creating a functional, user friendly, multi platform interface. However, it should be noted that the system is fairly limited in its current being. The controller where this interface is connected to does not provide more functionality. Therefore, if this interface is to be extended, it can only do so when the controller is extended first.

Extra functionality that would be very useful to have includes:

- Modify schedule items
- Delete schedule items
- Reccuring schedule items
- Current energy prices
- Energy price forecast
- Combining several actions into tasks

The usability test described in chapter 6 shows that the prototype is intuitive, to both users that have never worked with the system before, and users that already have had an short introduction to the system. In short we can conclude the system is easy to use, and could perhaps only profit from small explanation on the scheduler page.

References

[1] Play framework - build modern & scalable web apps with java and scala. June 2013. URL `http://www.playframework.com/`.

[2] Redis. June 2013. URL `http://redis.io`.

[3] The scala programming language. June 2013. URL `http://www.scala-lang.org/`.

[4] Yevgeniy Brikman. The play framework at linkedin. June 2013. URL `http://engineering.linkedin.com/play/play-framework-linkedin`.

[5] GreenerBuildings consortium. Approach. June 2013. URL `http://greenerbuildings.eu/approach`.

[6] Diane Cook and Sajal Das. *Smart Environments: Technology, Protocols and Applications (Wiley Series on Parallel and Distributed Computing)*. Wiley-Interscience, 2004. ISBN 0471544485.

[7] Georgievski et al. Optimizing offices for the smart grid. November 2011.

[8] R. Fielding. Architectural styles and the design of network-based software architectures. 2000. URL `http://www.ics.uci.edu/~fielding/pubs/dissertation/top.htm`.

[9] Martin Odersky. What is scala? June 2013. URL `http://www.scala-lang.org/what-is-scala.html`.

[10] Energy Smart Offices Project. Approach. June 2013. URL `http://www.ensoffices.nl/index.php/approach`.

A Redispush source code

Redispush is an application used for simulating a smart office where live data is pushed to a redis server. This data is fetched using the REST service, and pushes it the redis server. It depends on the freely available hiredis library.

A.1 redispush.c

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <curl/curl.h>

#include "hiredis.h"

void download_profile(void);

struct MemoryStruct {
    char *memory;
    size_t size;
};

static struct MemoryStruct chunk;

int main(void) {
    unsigned int j;
    redisContext *c;
    redisReply *reply;

    /* 1.5 sec timeout */
    struct timeval timeout = { 1, 500000 };
    c = redisConnectWithTimeout((char*)"127.0.0.1", 6379, timeout);
    if (c == NULL || c->err) {
        if (c) {
            fprintf(stderr, "Connection error: %s\n", c->errstr);
            redisFree(c);
        } else {
            fprintf(stderr, "Connection error: can't allocate redis con
        }
        exit(1);
    }

    while(1) {
        chunk.memory = malloc(1);
        chunk.size = 0;
        chunk.memory[0] = 0;

        download_profile();

        if(chunk.memory) {
            if(chunk.size > 0) {
                reply = redisCommand(c,"PUBLISH devicedata %s", chunk.m
                freeReplyObject(reply);
            }
            free(chunk.memory);
        }

        sleep(3);
    }

    return 0;
}


static size_t WriteMemoryCallback(void *contents, size_t size, size_t nm
    size_t realsize = size * nmemb;
    struct MemoryStruct *mem = (struct MemoryStruct *)userp;

    mem->memory = realloc(mem->memory, mem->size + realsize + 1);
    if(mem->memory == NULL) {
        fprintf(stderr, "Out of mem!\n");
        return 0;
    }

    memcpy(&(mem->memory[mem->size]), contents, realsize);
    mem->size += realsize;
    mem->memory[mem->size] = 0;

    return realsize;
}


void download_profile(void)
{
    CURL *curl_handle;
    CURLcode res;

    curl_global_init(CURL_GLOBAL_ALL);
    curl_handle = curl_easy_init();
    if(curl_handle) {
        curl_easy_setopt(curl_handle, CURLOPT_URL, "http://129.125.51.62
        curl_easy_setopt(curl_handle, CURLOPT_WRITEFUNCTION, WriteMemor
        curl_easy_setopt(curl_handle, CURLOPT_WRITEDATA, (void *)&chunk
        //curl_easy_setopt(curl_handle, CURLOPT_USERAGENT, "libcurl-agen

        res = curl_easy_perform(curl_handle);
        if(res != CURLE_OK)
            fprintf(stderr, "Curl failed: %s\n", curl_easy_strerror(res

        curl_easy_cleanup(curl_handle);
    }
    curl_global_cleanup();
}
```