a Green Mind project

Ruurtjan Pul (r.pul.1@student.rug.nl)
Brian Setz (b.setz@student.rug.nl)

12th of July 2013, Groningen

Table of Contents

1. Introduction 1
2. Architecture 2
   2.1 Global Architecture 2
   2.2 Client Architecture 5
   2.3 Server Architecture 7
      2.3.1 Integration with other components 8
3. Implementation 11
   3.1 Client Implementation 11
      3.1.1 Linux Implementation 11
      3.1.2 Windows Implementation 12
   3.2 Server Implementation 13
4. Compilation 15
   4.1 Compiling for Linux 15
   4.2 Compiling for Windows 16
5. Installation and Usage 17
   5.1 Sleepy Linux Installation 17
   5.2 Sleepy Windows Installation 17
   5.3 Sleep Management Server Installation 17
   5.4 Sleepy Usage 18
   5.5 Sleep Management Server Usage 19
6. Future Improvements 20
   6.1 Sleep proxy 20
   6.2 ZooKeeper integration 20
   6.3 Unique identifiers for workstations 21
   6.4 Debian package 21
   6.5 SSL 21
   6.6 Hibernate 21
   6.7 Message Broker 21
   6.8 Authentication & Authorization 21
7. Reflection 22

1. Introduction

Lazy Sleep has been development as part of the research internship done by Ruurtjan Pul and Brian Setz. It is part of the Green Mind project which focuses on increasing sustainability inside the Bernoulliborg, one of the buildings of the University of Groningen. For Green Mind to achieve this goal, multiple subprojects are being worked on by other students. Lazy Sleep is one of these projects.

The goal of Lazy Sleep is to minimize the energy usage of PC's (workstations) in the Bernoulliborg. The way Lazy Sleep achieves this goal is by taking control of the process for putting workstations into sleep mode. By doing this it is able to put workstations to sleep when no activity is detected and provide the administrators with important information with regards to the activity history of a workstation and whether the workstation should be sleeping or not.

In chapter 2 the architecture of the project is described. Followed by an explanation of implementation and design choices in chapter 3. Chapter 4 will explain the compilation process for both Windows and Linux. In the next chapter, chapter 5, the installation and usage of the software is described. A list of improvement that can be made in the future can be found in chapter 6. The report ends with a reflection in chapter 7.

2. Architecture

In this chapter the architecture of the Lazy Sleep project will be explained. In the first section a global overview of the architecture will be given in which each component will be discussed. Section 2.2 will contain a more detailed description of the architecture of the Sleepy client component of Lazy Sleep followed by the explanation of the Sleep Management Server in section 2.3.

2.1 Global Architecture

Figure 2.1 displays the overview of the entire architecture, including all the components that are involved as well as the flow of information between these components. These components can be divided into two separate groups:

- Components that will be developed by the Lazy Sleep development team
- Components which are developed by other developers of the Green Mind project and that have to be integrated with.

The architecture consists of the following components which are to be developed by the Lazy Sleep development team:

- **Sleepy Client**, cross platform client monitoring power management events and workstation activity while listening to and executing commands from the server.
- **Sleep Management Server**, cross platform server managing data received from clients and listening to requests from other components, forwarding these requests to the clients.

The architecture also consists of the following components that are developed by other developers of the Green Mind project and are used for integration purposes:

- **Database Server**, stores client specific settings as well as historical data.
- **Kafka Server**, activity data and power management events are sent to Kafka after being collected by the SMS, so other components can handle these events and data.
- **Orchestrator**, sends automated requests to the SMS, these are scheduled tasks.
- **Dashboard Web Server**, sends manual commands to the SMS.

What follows is a short description an example scenario demonstrating how these components interact. When a workstation starts, the Sleepy client will also start. The first thing the client does is retrieve the settings for this workstation from the Sleep Management Server (SMS) using TCP/IP. It will enforce the timeout before going to sleep that has been retrieved from the SMS. In the meanwhile it monitors activity of the workstation based on mouse and keyboard activity, this data is sent periodically to the SMS over TCP/IP. The SMS in turn sends this data to Kafka in order for it to be parsed and stored by an external component. Commands can be sent to the workstation by the Orchestrator (automated and scheduled tasks) or the Dashboard Web Server (manual commands) using the REST API of the SMS. When a command is received the SMS will make sure it will be forwarded to the correct workstation and executed on there.

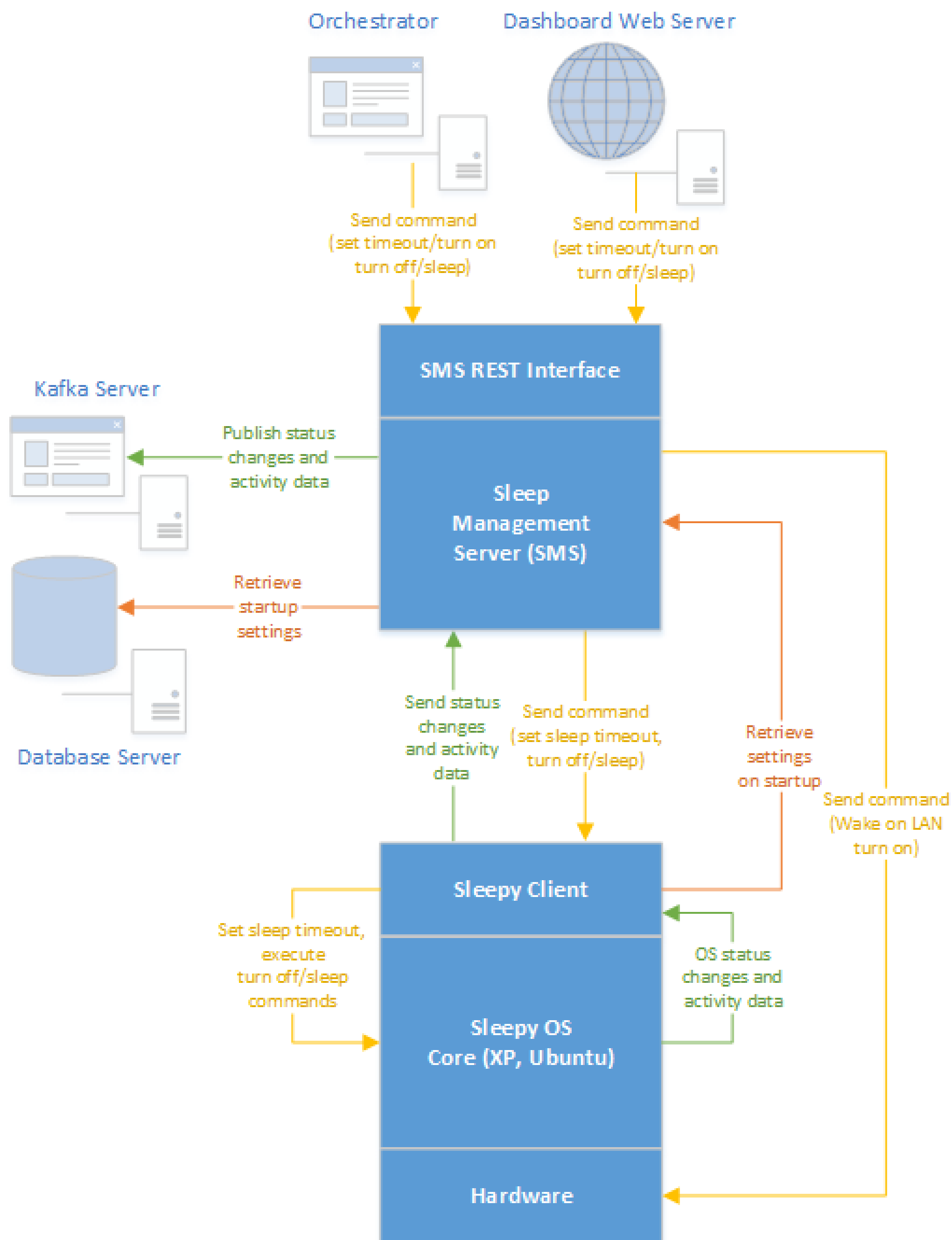

Figure 2.1: Global Architecture

The following tables will explain each component in detail. For each component their responsibilities will be describes, as well as the interface they provide to be accessed by other entities. Furthermore the dependencies of the component are listed and the developers who are responsible for developing the component. Only the components that are developed by the Lazy Sleep development team are described.

Component Name	Sleepy Client
Responsibilities	● Listen to commands (sleep/turn off/set timeout) from the Sleep Management Server. ● Listen to sleep events from the OS. ● Monitor activity on the workstation. ● Sleep or turn off the workstation based on activity or commands. ● Send power management events to the Sleep Management Server and Sleep Proxy Server when workstation turns off/enters sleep mode/wakes up. ● Send status (sleeping/awake/idle) and activity data to the Sleep Management Server. ● Retrieve client specific settings (e.g. time out before going to sleep) on startup. ● Change the timeout before a workstation enters sleep mode.
Interface	● Has a socket interface to receive commands from the Sleep Management Server. ● Uses the socket interface of the Sleep Management Server to send status and activity data to.
Dependencies	● Depends on the Sleep Management Server to retrieve settings from and send data and events to, as well as to receive commands from. ● Depends on the OS for implementation details such as sleep events.
Developers	Ruurtjan Pul & Brian Setz

Table 2.1 - Sleepy Client Component

Component Name	Sleep Management Server
Responsibilities	● Listen to commands (change settings/turn off/sleep/wake up) from the Dashboard Web Server. ● Listen to commands (status and activity data, request timeout settings) from the Sleepy Client ● Send commands (change settings/sleep/turn off) to the Sleepy Client. ● Send turn on command (Wake on LAN) to the workstation. ● Store data and settings changes received from the Dashboard Web Server and the Sleepy Client in the database.
Interface	● Has a REST interface to receive commands from the Dashboard Web Server. ● Has a socket interface to receive commands from the Sleepy Client. ● Uses the socket interface of the Sleepy Client to send commands (change settings/sleep/turn off). ● Uses the NIC interface of the workstation to send Wake on LAN.

Dependencies	- Depends on the Dashboard Web Server for receiving commands. - Depends on the Database Server for storing and retrieving client settings. - Depends on the Kafka for broadcasting events and activity of workstations. - Depends on the Orchestrator for fine grained control of workstations. - Depends on the Sleepy Client for activity and status data and for sending sleep/turn off/set timeout commands to. - Depends on the workstations for Wake on LAN support.
Developers	Ruurtjan Pul & Brian Setz

Table 2.2 - Sleep Management Server Component

2.2 Client Architecture

One of the requirements for Sleepy is that it should support Windows XP, Ubuntu with Gnome 2+, and Kubuntu with KDE, since these operating systems are supported by the RuG. Some features of sleepy however, are very dependent on the operating system or Linux display manager. To facilitate this, we separated the client into two categories:

- Sleepy core components
- Operating specific components.

Figure 2.2 shows the different components of Sleepy. The OS Core is an interface for all the operating system specific functionality. On startup, the corresponding operating system core is created in order for the Sleepy client to be able to call functions without knowing the operating system. The Settings Manager is used for retrieving run-time settings used to change the behaviour of Sleepy. The TCP / IP Server listens for connections of the Sleep Management Server. This connection is used to forward power management commands to Sleepy. Whenever the power state of a workstation changes, an event is published by the Event Publisher. The Activity Publisher sends the current activity status of a workstation periodically.

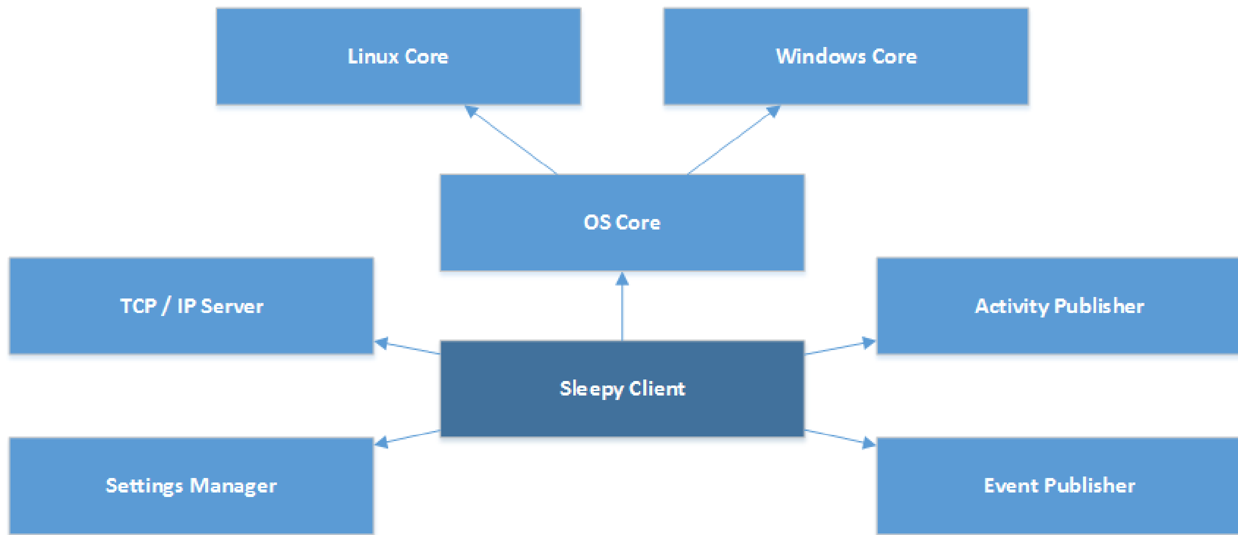

Figure 2.2 - Sleepy Client components

The Sleepy core, which can be seen in figure 2.3, takes care of operating system independent tasks like sending data to, and receiving commands from the Sleep Management Server. This is data about the activity of the user (e.g. the user has been inactive for 5 minutes), as well as power management events (e.g. the workstation went to sleep or is currently awake).

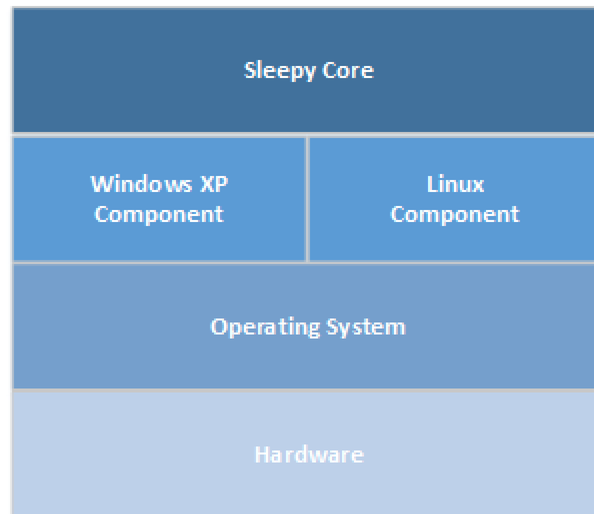

Figure 2.3 - The layered architecture of Sleepy

The activity is defined as the mouse and keyboard activity in the past 5 minutes. If there has been keyboard and mouse activity the past 5 minutes then the workstation is considered to be active. If there hasn't been any activity in the past 5 minutes then the workstation is considered to be idle. This information is sent every 5 minutes to the Sleep Management Server, which stores it in the database. The connection is realised via IP sockets; the Sleepy client initializes a connection whereas the Sleep Management Server listens to incoming connections.

When, in the dashboard, a user indicates that a workstation should be shut down, a command is sent from the Sleep Management Server to the Sleepy Core. The Sleepy core in turn accesses the operating system specific code required to shutdown the system. This code sends a shutdown command to the operating system. Figure 2.4 shows these inner workings of the Sleepy core.

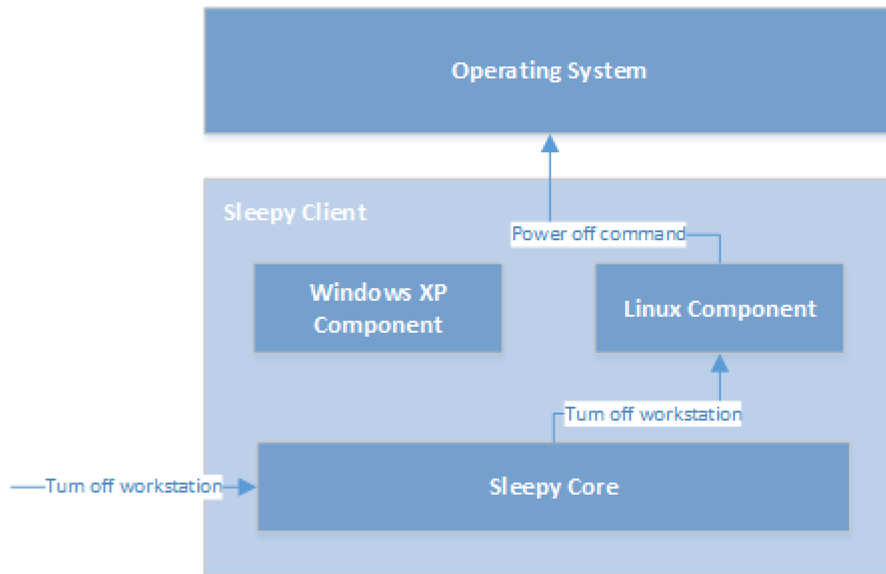

Figure 2.4 - Operating specific commands

2.3 Server Architecture

The Sleep Management Server can be divided into multiple sub components, each with their own role and responsibilities. Figure 2.5 shows the main components which make up the SMS.

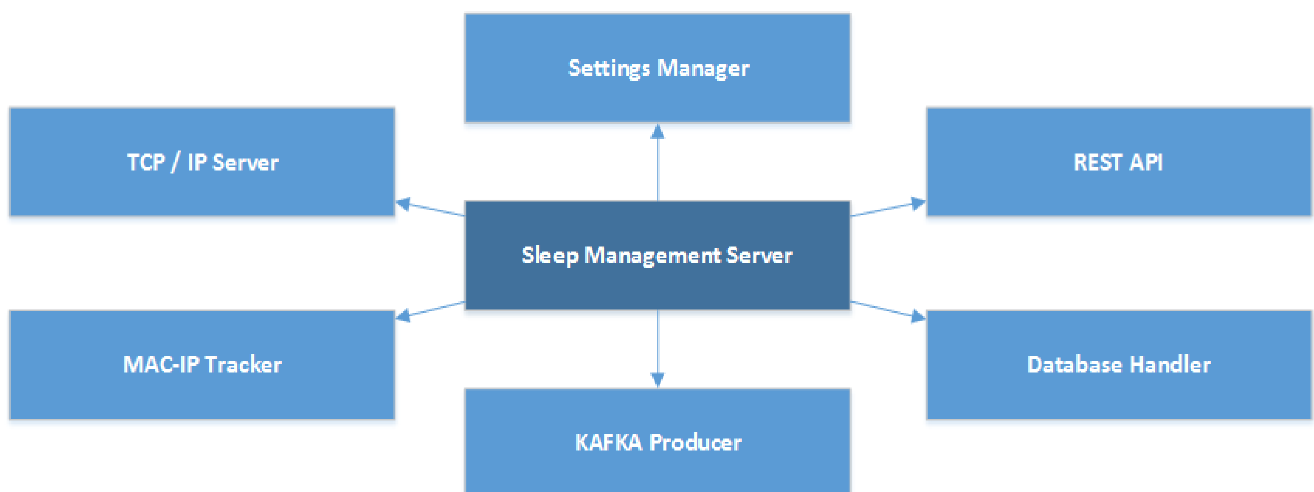

Figure 2.5 - Sleep Management Server components

- **Settings Manager,** is responsible for making application wide settings available to all other sub components. These settings are loaded from an .ini file.
- **REST API**, creates a HTTP server and sets up a REST API for the Orchestrator and Dashboard Web Server.
- **Database Handler**, handles all communication with the database and also provides easy ways to store and retrieve information to and from the database.
- **Kafka Producer**, pushes information about clients (power management events and activity status) to the Kafka server for other components to parse.
- **MAC-IP Tracker**, maps MAC-addresses to IP-addresses.
- **TCP / IP Server**, listens to connections from clients for settings request, power management events and activity data.

The Sleep Management Server functions mostly as a portal for other components to connect to and to translate messages from one component to messages that can be understood by another component. For example HTTP POST requests on the REST API will be translated to TCP/IP packets which can be understood by the Sleepy clients. Also, activity updates sent by Sleepy client to the Sleep Management Server are translated into messages that are pushed to Kafka where they are parsed and handled by other components.

2.3.1 Integration with other components

Three of these components integrate with other Green Mind projects, these components are the REST API, Kafka Producer and the Database handler. Because of these dependencies it is important to define the way the SMS interacts with these external components developed by other projects.

REST API

The REST API allows for external components to execute commands on workstations. These commands are:

- Change the time before going to sleep of a workstation.
- Send a workstation to sleep.
- Turn off a workstation.
- Wake up a workstation.

To achieve this the following REST API has been designed:

http://<host>:<port>/workstations/<mac-address>

- **<host>**: The hostname or IP of the Sleep Management Server.
- **<port>**: The port on which the Sleep Managment Server REST API is listening. The default port number is 40002 .
- **<mac-address>**: The MAC-address of the workstation to send this command to.

The HTTP requests send to this API have at least one of the two following POST data:

status=**<status>**
timeout=**<sleep timeout>**

- **<status>**: Possible values are: on,s3,s4,off. This changes the status of a workstation: turning it on, putting it to sleep or turning it off.
- **<sleep timeout>**: Set this field to change the time (in seconds) after which the workstation should go to sleep.

When an API request succeeds the following JSON response is given:

```
{
        status: "success"
}
```

When API request fails the following JSON response is given:

```
{
        status: "error",
        message: "some error message"
}
```

Note that the message is optional is not always returned. An example of a REST API request to shutdown a workstation with MAC-address 00:24:21:6F:01:24 would be:

URL: http://192.168.1.1:40002/workstations/0024216F0142
POST data: status=off

An example of a REST API request to change the sleep timeout of a workstation with MAC-address 00:24:21:6F:01:24 to two minutes (120 seconds) would be:

URL: http://192.168.1.1:40002/workstations/0024216F0142
POST data: timeout=120

Kafka Producer
The Kafka producer in the SMS collects all activity data and power management events from workstations and pushes these to the server. The format of these messages are as follows:

<mac-address>:activity:<activity-status>
<mac-address>:event:<event-type>

- **<mac-address>**: The MAC-address of the workstation of which this message originates from.
- **<activity-status>**: Possible values are: idle, active. Determines the activity of the workstation.
- **<event-type>**: Possible values are: standby,shutdown,turnedon. Describes the event type that occurred.

An example of a Kafka message showing that a workstation with MAC-address is 00:24:21:6F:01:24 is active would be:

0024216F0142:activity:active

An example of Kafka message showing that a workstation with MAC-address is 00:24:21:6F:01:24 has just turned on would be:

0024216F0142:event:turnedon

Database
The database is part of a separate project within Green Mind. We have provided our database design to the developers working on the database in order for them to implement it for us. This database design can be seen in figure 2.6.

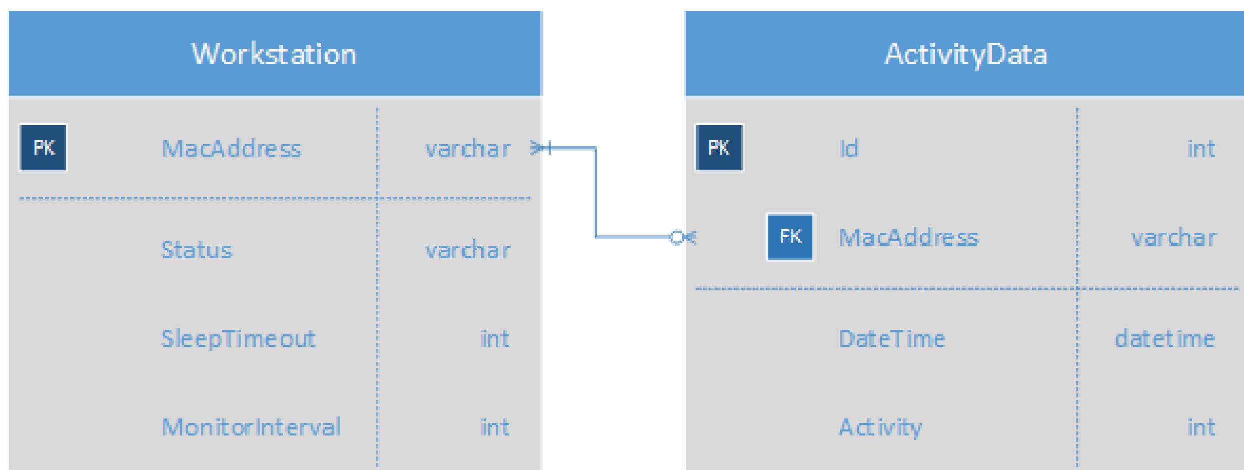

Figure 2.6 - Database design

3. Implementation

This chapter describes the implementation of the Lazy Sleep project as well as some important design decisions. Section 3.1 describes the client implementation. The server implementation is described in section 3.2.

3.1 Client Implementation

The design decisions for both the Linux and the Windows implementation of the client are described in this section. For each platform, the implementation details of the following aspects are described: Start on boot, sleep timeout, power management events, activity monitoring, shutdown and sleep.

3.1.1 Linux Implementation

This subsection describes the implementation details of the Linux sleepy client.

Start on boot

In Linux, there is more than one way to automatically start a program at boot time. We have analysed and tried several of these. One way is to add a start command to the '/etc/rc.local'. This script is run at the end of the OS boot, right before the display manager is started. Since no display manager is started yet, anything run at that point will not be able to connect to the Xserver started by this display manager.

Another way to start Sleepy at boot time is to use one of the display manager's scripts. The three scripts are: Xsetup, Xstartup and Xsession. These scripts are run at different times during the boot sequence. Since we want Sleepy to run even if no one is logged in, we decided to use Xsetup. This script is available for both kdm3 and gdm3.

Sleep timeout

There is no native functionality in Linux that puts a workstation to sleep after a period of inactivity. Most window managers have implemented this feature, but they are all specific to this window manager. In order to manage this independent of the window manager, there were two options applicable for our project:

1. xautolock[1]
2. sleepd[2]

Xautolock is a more generic program, since it is able to execute any script after an idle timeout. Sleepd is written specifically for putting a system to sleep and seems to be more mature. Furthermore, xautolock has a maximum timeout of one hour. This is why we chose to use sleepd instead.

---

[1] http://linux.die.net/man/1/xautolock

[2] http://joeyh.name/code/sleepd/

Power management events
Contrary to automatic starting of programs, catching power management events do not rely on an implementation of the display manager. Catching shutdown and restart events can be done by adding a script to '/etc/rc0.d' and '/etc/rc6.d' respectively, these scripts are executed on shutdown and reboot. Scripts in the '/etc/sleep.d/' folder will be executed when the system goes to S3 and S4 sleep and when it wakes up from these states.

Activity Monitoring
In order to monitor activity, the existing package 'xprintidle' is used. When xprintidle is executed, it prints the time a user has been idle. This package is available in the debian repository and was therefore chosen to monitor activity.

Shutdown & Sleep
To shut the workstation down, the command 'shutdown -h now' was used. Sleeping was done by executing 'pm-suspend'.

3.1.2 Windows Implementation

This subsection describes the implementation details of the Windows sleepy client.

Start on boot
To start a Windows application on boot it is best practice to turn this application into a service. This means the following has to be implemented in the application:

- A Main Entry point
- A Service Entry point
- A Service Control Handler

The Main Entry Point is called when a service is started in Windows. It serves the same purpose as a normal Entry Point for C/C++ application, with the difference being that the Main Entry Point of a service also creates a service table entry and executes the Service Entry Point.

The purpose of the Service Entry Point is to initialize all variables required to start a Windows Service. The status of the service is also managed by the Service Entry Point. As well as executing startup procedures, for example starting a thread which runs your code. It also registers the Service Control Handler.

In the Service Control Handler events from service are handled. It will listen to messages such as SERVICE_CONTROL_STOP requests which are sent when the service needs to stop. There also other events that are handled here such a pause and shutdown events.

Sleep timeout
In order to change the sleep timeout on Windows the powercfg.exe tool is used. This tool controls power settings and configures whether to go to Hibernate or Standby. In the Sleepy client this tool is called, it creates a new schema and changes the timeout before going to sleep to the requested timeout in seconds. Then it changes the active schema to this newly created schema.

The only difficulty with powercfg.exe is that the implementation varies on different Windows version. For example on Windows NT5.x (Windows XP/2000) the arguments of the commands that are used are different from the ones used on Windows NT6.x (Windows Vista, 7, 8). Therefore a check had to be implemented in to determine which arguments and commands to call based on the Windows version the client is running on.

Power management events

In Windows events are broadcasted when power management changes occur. These events are called WM_POWERBROADCAST events. We are interested in two subtypes of these events: EVENT_TYPE_SLEEP for when a workstation is entering sleep mode and EVENT_TYPE_TURNEDON when a computer is turned on. To detect shutdown events the application listens to WM_ENDSESSION with the event type being the following: EVENT_TYPE_SHUTDOWN.

Activity Monitoring

The activity is monitored using Windows Hooks. Both mouse and keyboard hooks are installed when the application is running. They listen to keyboard presses (any key press) and mouse movement. As soon as this movement is detected the workstation is marked as active until the activity is sent to the Sleep Management Server. When the activity is sent, the workstation will be marked as idle till there is mouse or keyboard activity.

Shutdown & Sleep

Putting a workstation into sleep mode is done by calling the SetSuspendState function that is part of the powrprof library of Windows. Depending on the arguments of this function the workstation will either go into hibernation or suspend to ram.

In order to shutdown a workstation the InitiateSystemShutdown function is called. However, in order to call this function elevated privileges are required, therefore these have to be requested before being allowed to call this function.

3.2 Server Implementation

This section describes the design decisions of the sleep management server.

Stateful vs Stateless

The Sleep Management Server is stateless, it does not care about the state of a workstation. The advantage of this is that clients can connect and disconnect at will without causing any problems in the SMS. Another advantage is that if the server goes down it can be restarted without having the reinitialize all clients.

The disadvantage is that more advanced commands that use the status of a workstation cannot be executed by the server, but have to be executed one level higher. This means an application that wants to execute said advanced command will have to query the database to find out the status of a workstation and then translate this information to the simple commands that the SMS can execute.

Sockets vs Message Broker

Early in the design process the decision had to be made whether to use sockets or message broker to communicate from the server to the client. The advantage of using sockets is that it is fast and easy to setup. Also, there is no dependency on third party software when the client and server are communicating.

The advantage of using a message broker to push commands to the clients is that is a lot more secure than sockets. The disadvantage is that it is time consuming the setup and configure and it also requires third party libraries on both client and server.

Because of the limited time available for this project it was decided to use sockets instead of the message broker but take into account that this shall be changed at some point. During the implementation the socket communication has been decoupled from the rest of the application, making it easier to replace with message broker functionality.

4. Compilation

This chapter describes the compilation process for both Linux and Windows. Compilation is described for Eclipse using the source code from the Git repository.

4.1 Compiling for Linux

To compile the Lazy Sleep project for Linux, follow these steps:

- Install packaged dependencies:
    - 'sudo apt-get install g++ cmake subversion eclipse-cdt libsqlite3-dev xprintidle sleepd'
- Install libmicrohttpd:
    - 'cd ~; wget http://ftp.gnu.org/gnu/libmicrohttpd/libmicrohttpd-0.9.27.tar.gz; tar -zxf libmicrohttpd-0.9.27.tar.gz; cd libmicrohttpd-0.9.27; ./configure; make; sudo make install'
- Install soci:
    - download and unzip soci: http://soci.sourceforge.net/. 'cd soci-3.2.1/cmake; cmake -G "Unix Makefiles" -DWITH_SQLITE3=ON -DSOCI_STATIC=ON ../; make; sudo make install'
- Install boost:
    - Download boost _1_53_0, boost-log-2.0 and threadpool-0_2_5-src
    - Unzip boost. Unzip boost log and threadpool into the unzipped boost folder.
    - Open the \include\boost-1_53\boost\threadpool\task_adaptors.hpp file and rename 'TIME_UTC' to 'TIME_UTC' with an underscore at the end
    - Configure boost build environment: './bootstrap.sh gcc --with-libraries=program_options,filesystem,system,thread,log'
    - Compile boost: './bjam --prefix=/home/boost32 toolset=gcc address-model=32 variant=debug link=static threading=multi install'
- Start Eclipse
- Checkout the source code: 'cd ~/workspace; svn co http://sm4all-project.eu/greenmind.svn/trunk/lazysleep/'
- Copy the cproject files: 'cp lazysleep/SleepyCore/cproject/linux.cproject lazysleep/SleepyCore/.cproject; cp lazysleep/SleepManagementServer/cproject/linux.cproject lazysleep/SleepManagementServer/.cproject; cp lazysleep/LazySleepLibrary/cproject/linux.cproject lazysleep/LazySleepLibrary/.cproject'
- In Eclipse: 'file -> import -> general -> existing project into workspace'. Browse to and select the projects.
- Double check the configuration paths! 'project explorer' right click on project -> 'Properties' -> 'C/C++ Build' -> 'Settings' -> 'Tool Settings' -> 'GCC C++ Compiler / Include' and 'GCC C++ Linker / Libraries'
- Build the projects 'ctrl + b'
    - If any problems occur, be sure to check the project settings. 'Project -> clean', and 'index -> rebuild' if you make any changes to libraries (installing, changing location etc.).

Before running the Sleep Management Server and Sleepy, check the configuration files. A description of these files can be found in the next chapter.

4.2 Compiling for Windows

To compile the Lazy Sleep project for Windows, follow these steps in order to compile the client:

- Install compiler toolchain
  - Download MinGW (http://sourceforge.net/projects/mingw/files/latest/download?source=files)
    - Install in C:\MinGW
    - Check option C++ compiler
    - Download ml.exe (http://www.microsoft.com/en-us/download/details.aspx?id=12654)
    - Download mc.exe (Visual Studio)
- Compile Boost 1.54.0
  - Download Boost (http://sourceforge.net/projects/boost/files/latest/download)
    - Extract in C:\MinGW
  - Download Boost Log 2.0 (http://sourceforge.net/projects/boost-log/files/latest/download)
    - Extract in 'boost' and 'lib' folder from boost-log-2.0 to C:\MinGW\boost_1_54_0
  - Download Boost Theadpool 0.2.5 (http://sourceforge.net/projects/threadpool/files/latest/download)
    - Extract in 'boost' and 'lib' folder from threadpool to C:\MinGW\boost_1_54_0
  - Open command line in boost_1_54_0
    - Execute: bootstrap.bat mingw --with-libraries=program_options,filesystem,system,thread,log
    - Execute: bjam.exe --prefix=c:\mingw\boost32 toolset=gcc address-model=32 variant=debug link=static threading=multi runtime-link=static define=BOOST_LOG_NO_COMPILER_TLS install
  - Open the file C:\MinGW\boost32\include\boost-1_54\boost\threadpool\task_adaptors.hpp and change TIME_UTC to TIME_UTC_
- Build Sleepy client
  - Install Eclipse CDT (http://www.eclipse.org/cdt/downloads.php)
  - Obtain Lazy Sleep source code
    - SVN: http://sm4all-project.eu/greenmind.svn/trunk/lazysleep/
  - Move the windows.cproject file from SleepyCore/cproject/ to SleepyCore/ and rename it to .cproject
  - Move the windows.cproject file from LazySleepLibrary/cproject/ to LazySleepLibrary/ and rename it to .cproject
  - Open Eclipse, import projects
  - Press CTRL+B

5. Installation and Usage

Information about the deployment of Lazy Sleep can be found in this chapter. The first and the second section describe the installation of Sleepy on Linux and Windows respectively. Section 5.3 provides information about the installation of the Sleep Management Server. The usage of Sleepy is described in section 5.4, while the last section describes the usage of the Sleep Management Server.

5.1 Sleepy Linux Installation

In order to use Sleepy on Linux, copy all the required files and execute install.sh to install them to the correct location. This script also installs the required dependencies. Manually enable wake-on-lan by adding 'ethtool -s eth0 wol g' to '/etc/rc.local' and configuring the BIOS. To automatically start Sleepy, add it to the display manager's script as described in install.sh's comments.

5.2 Sleepy Windows Installation

To use the Sleepy client on Windows it has to be installed as a service. Included with the software is an install.bat script which contains the commands to configure and install the Sleepy client as a service using the sc.exe utility that is part of Windows.

After the Sleepy client has been installed as a service the port that it listens on has to be opened in the firewall. The port that has to be opened is port 40003. When the port has been opened Sleepy is ready and can listen to commands from the server.

5.3 Sleep Management Server Installation

To install the Sleep Management Server, copy the binary and the configuration files to the server using Secure Copy. Start it by executing 'sudo ./SleepManagementServer'.

5.4 Sleepy Usage

What follows are the command line parameters of the Sleepy client including a description of each parameter.

Generic Options

Option	Description
--help	Produce help message.
-c [--config] arg (=sms_settings.ini)	Name of a file of a configuration.
-l [--logconfig] arg (=sms_logsettings.ini)	Name of a file of a logging configuration.

Configuration Options

Option	Description
--developer_mode arg (=0)	Set to true to skip startup checks.
--sms_host arg (=127.0.0.1)	The hostname or IP of SMS.
--sms_port arg (=40001)	The port SMS is listening on.
--linux_event_port arg (=40004)	The port the Linux Event server is listening on.
--sleepy_port arg (=40003)	The port sleepy starts listening on for commands from the SMS.
--connection_tries arg (=5)	The number of times a connection will be tried to be made (if the first time fails).
--connection_timeout arg (=5000)	The timeout in milliseconds between connection tries (if the first time fails).
--thread_pool_size arg (=10)	The maximum number of clients that can connect.
--network_adapter_name arg (=Ethernet)	The network adapter name.

5.5 Sleep Management Server Usage

What follows are the command line parameters of the Sleep Management Server including a description of each parameter.

Generic Options

Option	Description
--help	Produce help message.
-c [--config] arg (=sms_settings.ini)	Name of a file of a configuration.
-l [--logconfig] arg (=sms_logsettings.ini)	Name of a file of a logging configuration.

Configuration Options

Option	Description
--api_port arg (=40001)	The port SMS is listening on for incoming sleepy connections.
--sms_port arg (=40002)	The port SMS is listening on for incoming api calls.
--sleepy_port arg (=40003)	The port sleepy clients are listening on.
--wol_broadcast arg (=255.255.255.255)	The IP of the broadcast destination for Wake on LAN.
--wol_port arg (=9)	The port of the workstation for Wake on LAN.
--connection_tries arg (=5)	The number of times a connection will be tried to be made (if the first time fails).
--connection_timeout arg (=5000)	The timeout in milliseconds between connection tries (if the first time fails).
--thread_pool_size arg (=500)	The maximum number of clients that can connect.
--activity_timeout arg (=60000)	The default timeout between activity updates.
--sleep_timeout arg (=3600)	The default timeout in seconds before going to sleep.
--alivecheck_timeout arg (=10000)	The timeout between alive check on workstations.
--time_to_live arg (=60000)	The time before a workstations is considered to be offline.
--kafka_host arg (=127.0.0.1)	The host on which the kafka server runs.
--kafka_port arg (=9092)	The port on which the kafka server listens.
--kafka_topic arg (=LazySleep)	The kafka topic to publish messages to.

6. Future Improvements

In this chapter, we describe possible future improvements that could be done for the Lazy Sleep project. Some of these improvements were in the SCRUM backlog at the beginning of the project, while others were added during the project. But first we shall cover the issues that currently exists in the project

Waking up from sleep on Linux

Linux workstations have difficulty waking up from sleep mode. They hang as soon as they are woken up. It is very likely that this has something to do with NVIDIA graphics cards.

Wake on LAN

The implementation of Wake on LAN on the server has been tested but it does not function on the RuG network. The BIOS of workstations has been checked, yet they have Wake on LAN enabled. A possible reason for Wake on LAN not working is that Wake on LAN packets are blocked on the network.

Windows Service

There exists a bug on Windows where running the client as a service causes a segmentation fault. This has most likely something to do with Boost, either the Boost Log library or the Boost thread library. There are known issues on Windows XP for both these libraries.

In the following subchapters future improvements will be described.

6.1 Sleep proxy

One of our initial ideas for the Lazy Sleep project was implementing a sleep proxy as described by Joshua Reich[3] et. al. This sleep proxy is able to route network to itself when workstations are put to sleep. Whenever network traffic for a workstation is detected, the workstation is woken and the traffic is sent to the workstation. The advantage of this is that users do not have to send manual commands to wake up a workstation if they require access to it. Several implementations have been developed, but none of them suit our requirements and are open sourced. Due to the limited amount of time available to us, we decided to push the sleep proxy to the back of the project.

6.2 ZooKeeper integration

Halfway through the project, an integration camp was held to integrate the different components of the Green Minds project. In this integration camp, we decided to use ZooKeeper[4] to discover IP addresses of other components. This way, no configuration has to be hard-coded into the source code and do we achieve a uniform way of publishing this configuration. However, there are no mature, cross-platform client implementations of ZooKeeper for C++. This, in addition to the change being in the middle of the project and the limited available time, we were not able to

[3] Sleepless in Seattle No Longer

[4] http://zookeeper.apache.org/

implement this integration. ZooKeeper integration would be a great addition to the Lazy Sleep project to achieve better integration with the Green Minds project.

6.3 Unique identifiers for workstations

In the current implementation of the project workstations are identified using MAC addresses. In theory MAC addresses are unique, but it is possible to spoof these MAC addresses. Therefore it would be more secure to use a form of unique identifier for workstations instead of their MAC address.

6.4 Debian package

In order to deploy Sleepy on Linux machines in the Bernoulliborg, a Debian package should be made. This way it can be deployed using the existing deployment channels of LWP and OIC. At this point in the development process, the software is not mature enough to be deployed for the masses. This is why no Debian package has been made yet.

6.5 SSL

When using sockets to communicate from the Sleep Management Server to the Sleepy Client it is important to do this over a secure connection. Not everyone should be allowed to send commands to clients, therefore the source of the commands needs to be trusted and identifiable. This is also true for communication between HTTP clients and the REST API.

6.6 Hibernate

In addition to S3 sleep, S4 hibernate can be added to the system. The difference between S3 and S4 sleep is the way memory is stored. In S3 sleep, the memory is kept on RAM. This way waking up from sleep is a lot faster, but more energy is required. S4 hibernate the state is written to the hard disk. The advantage is that only the network card needs to be powered to be able to receive wake-on-lan messages. However, waking up takes longer.

6.7 Message Broker

See chapter 3.2 for more information about why using a message broker is preferable to using sockets. In short, using a message broker will greatly enhance the security on the client side.

6.8 Authentication & Authorization

Before an entity is allowed to use the REST API it should first be authenticated and authorized to execute the command. Therefore checks needs to be in place to do authentication and authorization when executing HTTP POST requests on the REST API.

7. Reflection

Now that we have finished the project we can look back at the development process and determine what we did right and what we could have done better. We shall start with things that could have gone better. After that we will discuss the things that we did right.

Defining and communicating with stakeholders is a something that could have gone better. In the beginning of the project Faris and Tuan acted as stakeholders of our project, they gave us requirements and provided us with good feedback about these requirements. However, as we neared the end of the project the LWP staff was introduced as stakeholders. Because they weren't involved since the beginning they had all kinds of functionality requests which were never mentioned by Faris and Tuan. This caused some difficulties because we did not have the time to incorporate the functionality they would have liked to see.

It would have been better to involve the LWP staff from the beginning of the project. Also, the same problem might arise when it comes to the people responsible for deploying Windows installations as we have not had contact with them whatsoever. In short, this could have gone much smoother if all stakeholders were known upfront and if we had talked to every stakeholder about their concerns and requirements.

One of the things that went great was the communication with Faris and Tuan. They were almost always available and willing to make and spend time working on and discussing the project. They also acted proactively in removing blocking issues that would have slowed the project down considerably.

Also, the decision on our part to use SCRUM as the development method was a good choice for this project because of the changing requirements and short development schedule. The result was that we were able to develop rapidly and implement a lot of functionality before the end of the project. It also meant that Faris and Tuan gained a lot of insight in our development process and were actively involved during demonstrations at the end of a SCRUM sprint cycle. The only downside was that when Faris and Tuan were unavailable for a SCRUM demonstration we had to do it by e-mail, this didn't work very well.

Each SCRUM sprint had the duration of one week. At the beginning of the week we decided, in consultation with Faris and Tuan, what SCRUM items were to be done the coming week. Each SCRUM item consist of a description, as well as an amount of points. The burn down charts in figure 7.1 show the progress of week one, two, three, and five. The horizontal axis show the progression of the week, whereas the vertical axis shows the number of remaining points. The first sprint, we planned to many items, which resulted in remaining points at the end of the sprint. To counter this, we planned less work, and adjusted the amount of points we assigned to each task. This time however, we planned to little work. We had to plan new items at the end of day two and three. After a while, we got used to the method and were able to plan sprints properly, as can be seen in the burn down charts of week three and five.

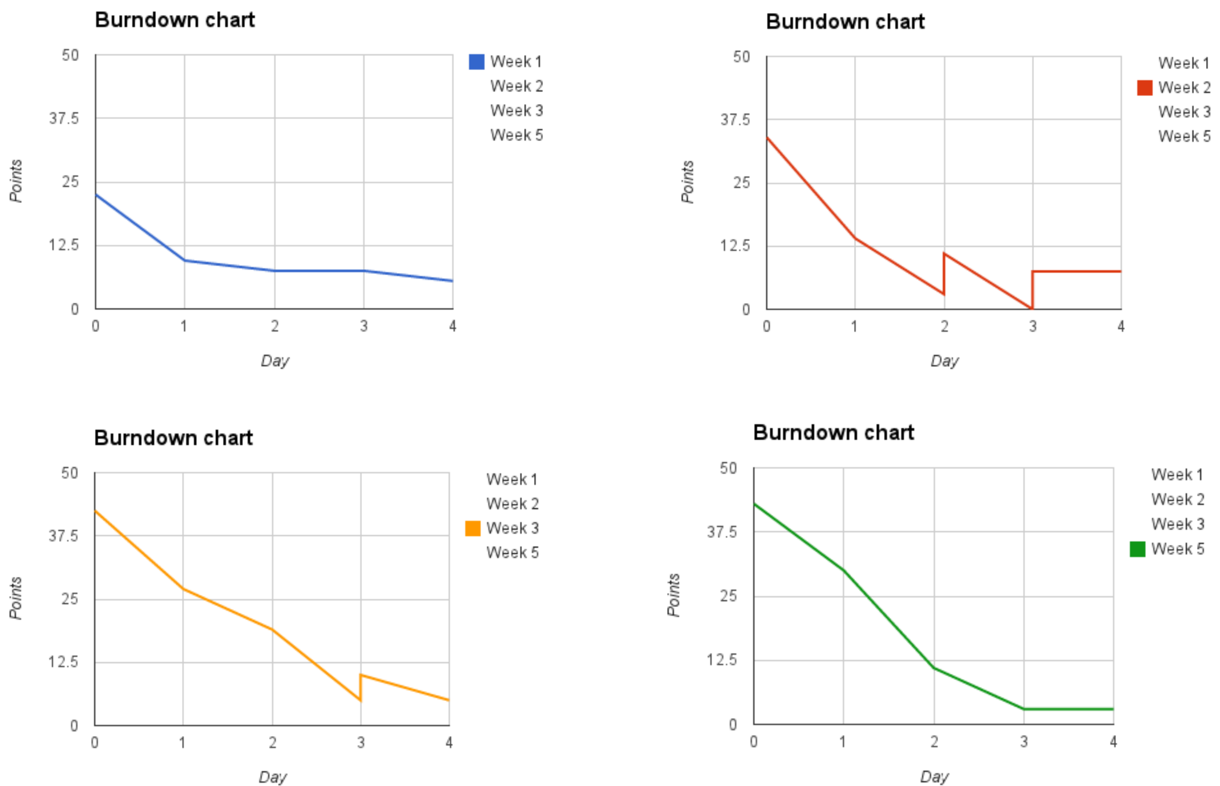

Figure 7.1: Burndown charts

The decision of using C++ as the language of choice for both the client and the server, and the decision to program cross-platform turned out to be both a good decision and a bad decision. The reason it was a good decision was because we had almost no experience with C++, this meant we learned a lot about programming in C++ during the two months we had. Also, neither of us had done any systems programming on Windows and Linux, so this was very interesting to do. The down side of our inexperience is that we made mistakes that could have been prevented if we had more experience with both C++ and system programming.

When looking back to how things have gone and the decisions we have made we can conclude that we are content with the results of the project. We have implemented a server, a Linux client and a Windows client in a programming we had no previous experience with. Yet we were able to implement all the major functionalities that were requested by Faris and Tuan. We also think that the architecture we have designed for Lazy Sleep is solid, the code itself however does need some work due to our inexperience with C++.

Therefore we think that our research internship was successful, we also think that our part of the Green Mind project has the potential to lower the energy usage of workstations and achieve the goal it was designed for: minimizing the energy usage of workstations in the Bernoulliborg.