www.FirstRanker.com

IT 10 029
Examensarbete 15 hp
Juni 2010

Demonstrating the Effects of Power Management on a Real-Time Operating System

Peter Backeman & Erik Gustafsson

Institutionen för informationsteknologi
Department of Information Technology

UPPSALA
UNIVERSITET

Teknisk- naturvetenskaplig fakultet
UTH-enheten

Besöksadress:
Ångströmlaboratoriet
Lägerhyddsvägen 1
Hus 4, Plan 0

Postadress:
Box 536
751 21 Uppsala

Telefon:
018 – 471 30 03

Telefax:
018 – 471 30 00

Hemsida:
http://www.teknat.uu.se/student

Abstract

Demonstrating the Effects of Power Management on a Real-Time Operating System

Peter Backeman & Erik Gustafsson

This thesis is part of the GEODES project, dealing with the issues of power optimization, e.g. how to make a systems lifetime longer, at the cost of the quality of the performance, this is called system degradation.

The objective for this project was develop an application to monitor and visualize the effects of power management (controlled system degradation to minimize power consumption). With such software it is possible to demonstrate the possibilities of so called energy aware systems.

During the project two scenarios were investigated and implemented, one is a simple state-based monitor. This monitor application can observe and visualize the effects of power management on a computer system. It shows that by utilizing system degradation, the lifetime of a system can be prolonged.

The other scenario consists of a prototype for a firefighter coordinator application. It allows a rescue leader to observe firefighters at an emergency scene. This application can interact with the power management, by requesting a desired lifetime of a hand-held device carried by the firefighter. This shows that these techniques can be utilized without knowledge of the underlying power management software.

Handledare: Barbro Claesson & Detlef Scholle
Ämnesgranskare: Karl Marklund
Examinator: Anders Jansson
IT 10 029
Tryckt av: Reprocentralen ITC

Sammanfattning

Detta examensarbete är en del i GEODES-projektet, som arbetar med energioptimering, t.ex. hur man får ett inbyggt systems batteri att leva längre. Detta kan göras på kostnad av kvaliteten av prestandad, det kallas för systemdegradering.

Målet med detta projekt var att utveckla en applikation för att monitorerea och visualisera effekterna av energiförvaltning (kontrollerad systemdegradering för att minimera energikonsumption). Med sådan mjukvara är det möjligt att demonstrera möjligheterna med s.k. energimedvetna system.

Under detta projekt så var två scenarion undersökta och implementerade. Det ena scenariot var en simpel monitor och det andra var en prototyp för en brandmanskoordineringsapplikation. Båda applikationerna utnyttjar energiförvaltning och kan visualisera resultaten.

List of Figures

Terms and Abbreviations

Term/Abbreviation	Description
GEODES	Global Energy Optimization for Distributed Embedded Systems
LINX	An inter-process and inter-system communication library for OSE and Linux
MVC	Model-View-Controller
OSE	Operating System Embedded
Power Data	Data that contains information about power management
Power Manager	Software that performs local power management on a computer system
Power Management	Management of power, dynamically adjusting components power consumption
SRP	Single Responsibility Principle

Table 1: List of Terms and Abbreviations

Contents

Chapter 1

Introduction

This bachelor thesis is performed at Enea[1] and strives to develop a demo application to demonstrate the effects of power management (methods to minimize power consumption) on a computer system.

The thesis is part of the ITEA2[2] project GEODES[3] (Global Energy Optimization for Distributed Embedded Systems - ITEA2∼07013[4]) that aims at developing design techniques, embedded software and accompanying tools to help embedded system lower their power consumption.

1.1 Background

In embedded systems, power consumption is of great concern and it is of great value to minimize it. In the GEODES project, techniques are developed to lower power consumption at the cost of lesser performance, this is called system degradation. This thesis will work on demonstrating the effects of power management.

1.2 Problem Description

The task presented is to design software applications that demonstrates the effects of power management, which is the distribution of power amongst components and utilization system degradation to minimize the total power consumption.

To fulfill this assignment, new applications will be developed and then integrated with existing software. Primarily it will be collaborating with a software application called Power Manager. This application is the software module responsible for the power management[5].

The purpose of the project is divided into two different scenarios; one consists of the monitoring a computer system, the other consists of monitoring firefighters at an emergency scene.

[1] http://www.enea.com/
[2] http://www.itea2.org/
[3] http://geodes.ict.tuwien.ac.at/
[4] http://www.itea2.org/public/project_leaflets/GEODES_profile_oct-08.pdf
[5] For more information about the Power Manager see [2]

Monitoring of a Computer System

In this scenario, a user wishes to monitor the effects of power management on a computer system. By using the computer system monitoring software, the user can in real time see the current power consumption of different components in the system.

If the user wishes to change the state (e.g. the desired lifetime) of the monitored system, certain parameters can be set through an interface, and the resulting change (e.g. adaption of the power manager) can be observed.

The computer system to be monitored will run Enea's operating system OSE (Operating System Embedded)[6] on a Freescale[7] i.MX31 ADS board (abbreviated MX31), and the computer system monitoring will run Linux on a PC (regular x86 system).

Monitoring Firefighters at an Emergency Scene

In this scenario, a rescue leader of a fire brigade can use the firefighter monitor software to monitor a group of firefighters at an emergency scene. With this software the status of the individual firefighters (e.g. position, outer temperature, etc.) can be observed. This data is recorded and distributed by a handheld device carried by each firefighter.

The leader is also able to request a desired lifetime (battery-wise) of each handheld device. This deadline can be changed during the mission, and the device will adapt by using power management techniques.

The software aiding the leader to monitor the firefighters will run Linux on a PC and the handheld devices will run Enea's operating system OSE on a Freescale MX31 board.

1.2.1 Purpose

The purpose of this thesis is to demonstrate the effects of power management. It will be done by developing software to show that the lifetime of an embedded system can be prolonged by the use of system degradation.

During the thesis, there will also be investigation of the needs in a developer's toolkit for applications developers. The results can then help in the design of the middleware such that it can be designed to be easy to use.

1.2.2 Method

The project will be approached using an agile method of software development. This means that the implementation will begin early (with only a partial design of the system complete) and there will be continuous evaluation and modification of the design.

The actual development will consist of "sprints". A sprint is a period of time where the focus of work is a few chosen tasks which shall be accomplished before the end of the sprint. In this project each sprint will last one week.

An agile method has been favored because of the tight time frame and the problem of anticipating the amount of time each task would take. With this

[6] http://www.enea.com/OSE
[7] http://www.freescale.com/

method, the design (and amount of functionality) can easily be adapted as the project progresses and adjusted accordingly[4].

1.2.3 Delimitations

The project will be performed during a time frame of ten weeks. Since an agile approach of development is used there will be only a short draft of the design in the beginning (about two weeks) and afterwars a series of phases consisting of iterative implementation and elaboration of the design.

The environment consists in both scenarios of running one part of the software on a multimedia Freescale MX31 ADS board, and the other on a Desktop PC. The MX31 shall be running Enea's real time operating system OSE and the PC shall be running a Linux distribution. The communication between the applications across the systems will be done via Ethernet.

Chapter 2

Structuring of the Applications

The structures of the two software applications are very similar. Both scenarios consist of one part running on a Freescale MX31 board using OSE and the other on a Linux PC. Despite the similarities, there are a few crucial differences, the primary being that in the firefighter scenario, multiple firefighters are to be monitored by a single coordinator.

2.1 Context of the applications

In the problem description, it is stated that two applications shall be running on OSE. This causes that this software must be written for OSE, and therefore can only utilize the functions of OSE. More specifically, LINX will be used for communication and the applications will be written as OSE modules.

2.1.1 OSE

OSE is a real-time operating system developed by Enea. It is a widely used embedded operating system, used in a third of all mobile phones sold worldwide[1]. An application on OSE consists of one or several processes, which is the smallest executing entity.

2.1.2 LINX

LINX is a protocol for IPC (Inter-Process Communication) between processes within the same system and also between processes on different systems. It is the default IPC used by Enea's OSE, it also exists as an open source implementation as LINX for Linux[2].

The principle of LINX is to use something called Signals that are sent to processes, addressed by PIDs. It is an reliable, in-order communication protocol, which means that all signals sent, are received eventually, and they will be received in the correct order.

[1] http://www.enea.com/OSE
[2] linxdoc/doc/index.html

To acquire the PID of a process, to send a signal, a system call named *hunt* is used. To hunt, the name (as a string) of the process is passed, and the return value is the corresponding PID (if found).

The hunt method finds all process on the same computer with no problems, but if one wants to hunt for a process on another system on needs to set up a LINX link to the target host. These links can be created automatically by using a program called linxdisc (LINX discovery daemon). Linxdisc is a background process that repeatedly broadcasts its existence, and listens for the existence of other LINX nodes. If it finds other LINX nodes, a link is created automatically.

Using LINX, a process can *attach* a remote process. Thie means that if the connection between the local and the attached remote process is lost, LINX will notify the local process by sending a signal. This feature is used by the Firepad and the Central applications.

LINX will be used since in OSE it is the default message passing method, therefore the communication protocols are easily implemented. However, it was also used as the protocol for communication across the systems (from OSE to Linux and v. v.). The reason for this is the simplicity of having the same communication method between all nodes (and not need to bother whether the target process was on the same system or another).

2.1.3 Power Manager and Power Data

On the MX31 board running OSE, there will exist an application called the *Power Manager*. This application is responsible for the power management, and is the primary thing to be observed. The Power Manager is communicating with different power aware components of the computer system, called PMCs (Power Manageable Component), and issuing orders on allowed power consumption. It also receives information from these components, about their current power consumption. This data (orders and information) are to be collected, and are collectively named *Power Data*.

2.2 Architecture of the System Monitor Application

In the computer system monitor scenario, there will be two applications. The first one, called the Monitor shall be on the MX31 board running OSE. The second one, called the Visualizer will reside on a PC running Linux.

This split is made to reduce the complexity of the applications. By structuring it as two applications, one which focus on monitoring and the other on visualizing the monitored data, the design conforms to the SRP (Single Responsibility Principle)[4].

The SRP states that every object should do one thing and that thing should only be done by that object only. By following this principle, the design of the each modules will be easier. For example if the interface for monitoring the Power Manager changes, only the Monitor application will have to adapt, whereas the Visualizer can remain unchanged.

2.2.1 Requirements

The primary objective of the Monitor is to survey and collect Power Data from the Power Manager.

By loading the Monitor application on the board it can utilize OSE's own IPC (inter-process communication) for monitoring, and this makes the supervision of the power management easier. Therefore, the Monitor application will be running on the monitored board as a single (or several, if parallelism is desired) of OSE processes.

Since there is no existing functionality in OSE for peeking at memory or signals passed between other processes, the monitoring will be done by requiring the Power Manager to explicitly send power data to the Monitor process. This is the first requirement of the Monitor:

REQ1-1: The Monitor shall survey the Power Manager and collect Power Data.

After the data has been gathered, the Monitor should propagate this to the Visualizer, where it will be displayed. This is the next requirement of the application:

REQ1-2: The Monitor shall be able to send Power Data to the Visualizer.

The Visualizers task is to collect the power data and visualize it. Letting it be run on another computer system makes it possible to monitor a system from a different location. Also it allows the use of functionality not yet ported for OSE (e.g. certain graphics). This can be formulated as the first requirement of the Visualizer application:

REQ1-3: The Visualizer shall display the Power Data received from the Monitor.

It follows from the architecture sketch that a connection must be established between the Monitor and the Visualizer. Since the transmission of data lies primarily in the interest of the Visualizer, it should be responsible of initiating the transfer of Power Data. This is formalized as:

REQ1-4: The Visualizer shall be able to establish a connection the Monitor.

The most interesting thing to monitor is changes in the power state (e.g. new demands on required lifetime) of the target system. To initiate such a change, it is necessary for the application to be able to set some parameters at the Power Manager. This is most conveniently done from the Visualizer (since that's where the output is displayed):

REQ1-5: The Visualizer shall be able to send commands to the monitored system.

By fulfilling all of these requirements, the application will meet the general task

of monitoring and demonstrating the effects of power management on a real time operating system.

2.2.2 Use Case - Monitoring a Computer System

In the setup depicted in Figure 2.1 the Visualizer shall locate and connect to the Monitor residing on the board that the user desires to observe. With an established connection, the Visualizer can request data from the Monitor.

When the Monitor receives a request it will start and forward all monitored data to the Visualizer. The Monitor itself displays no information on the output of the board it is running on. Instead all data is presented on the Linux PC's display by the Visualizer application.

When data is flowing to the Visualizer, it visualizes the data in the form of tables and diagrams in real-time. Also, the Visualizer has the capability to set certain parameters at the Monitored host. The first parameter is the required lifetime, which is the period of time the user wishes the system to remain running. The second parameter is the batter lifetime, which is the amount of energy left in the system. The battery lifetime can be set because in this scenario, the battery is simulated.

With the ability to change these parameters, the user can see how the Power Manager adapts to changes of the state.

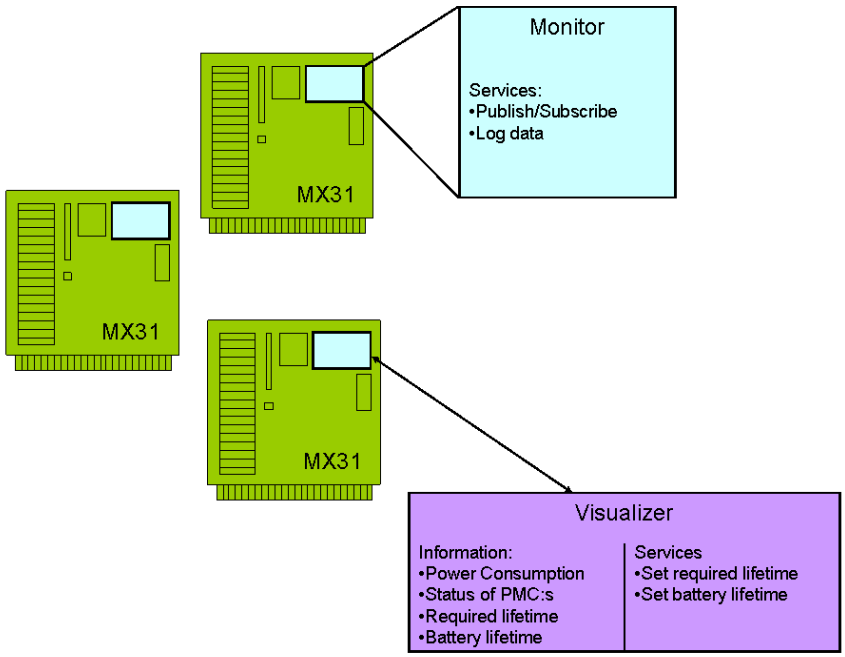

Figure 2.1: Monitoring a Computer System - Information presented and services available of different components in this scenario

2.3 Architecture of the Firefighter Application

The second scenario consists of a prototype for a central coordinator of firefighters as well as an application called Firepad which will run on handheld devices carried by the firefighters.

In this context, the division (the same division as in the computer system monitoring scenario, see Section 2.2) comes naturally, since the whole point of the application is to have the observer and the firefighters geographically apart (so the leader can survey all of the firefighters from the same location). Thus there will be two different applications, one used by the firefighters and one used by the rescue leader.

The application used on the handheld device carried by the firefighters is the *Firepad* application that will in our prototype run on the MX31 board.

The other application used by the rescue leader is called the *Central* which will run on the Linux PC.

An important aspect of this scenario, is that the Central shall have as little knowledge about Power Manager as possible, thus showing that power management can be used by applications almost unaware of the underlying software.

2.3.1 Requirements

The purpose of the Central is to survey the firefighters (or more specifically their Firepads) and display the received information about their current condition (e.g. position). This is the first requirement of the Central module:

REQ2-1: The Central shall display current condition of the connected Firepads.

In this scenario, as firefighters (with their Firepads) approaches the emergency scene, they should automatically be registered at the Central. This is formulated as the second requirement:

REQ2-2: A Firepad shall be able to automatically find the Central and connect to it.

When the Firepad is connected, it should regularly send information from its sensors to the Central. This capability is stated as:

REQ2-3: A Firepad shall be able to send data about its current condition to the Central.

A major point of this prototype is to show the capabilities of power management, and this is demonstrated by letting the Central have the capability to request a required lifetime of a Firepad. This request shall then be forwarded from the Firepad to the underlying power management software through a general API:

REQ2-4: The Central shall be able to set a required battery lifetime of a Firepad.

If these requirements are met, the firefighter application will demonstrate the ability to utilize power management without having a deep knowledge of the underlying power management software.

2.3.2 Use Case - Monitoring Firefighters

Figure 2.2 shows a typical use case scenario. In this setup there are three firefighters out in the field, each one carrying a Firepad. There is also a Central surveying an emergency scene. Whenever a firefighter enters the emergency scene, his or hers Firepad registers itself at the central.

After the registration is done, the Firepads sends various data including current position and temperature to the Central, which collects this and displays it on a screen. The coordinator (using the Central application) can monitor this and take proper action.

The coordinator also has the capability to use services of the Central software, which includes setting required lifetime of a Firepad (for example if the circumstances require the firefighter to stay on field for an extended period of time).

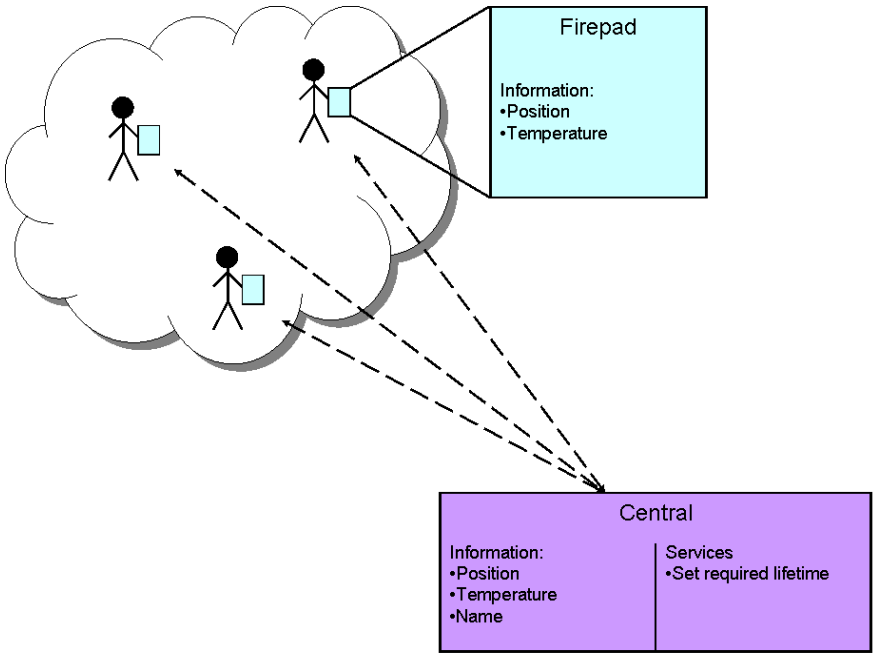

Figure 2.2: Monitoring Firefighters - In this scenario, the battery lifetime cannot be set, since that is not a realistic feature

Chapter 3

The Design of the Software

The software applications have been designed with the requirements in mind, making sure they are fulfilled. First the computer system monitor application was designed, and afterwards the firefighter application. Much of the design of the latter application resembles the design of the first. The design of the applications will be presented in the same order as they were written, but first follows an overview of the protocols used.

3.1 Protocols

The developed applications utilize a number of different protocols for communicating commands and data. Here is a presentation of the protocols for each scenario.

3.1.1 Protocols in the Monitor – Visualizer Scenario

In the Monitor – Visualizer scenario there are basically three kinds of communication:

Client	Server	Type
Visualizer	Monitor	Subscriptions
Power Manager	Monitor	Logging
Visualizer	Power Manager	Commands

Table 3.1: Protocols used in the Computer System Monitoring scenario

Monitor Visualizer, Subscriptions

The message passing between Visualizer and Monitor is based on the Publish/Subscribe design pattern (also known as the Observer Pattern[3]). This means that the Visualizer subscribes for data from the Monitor.

As seen in figure 3.1, the whole process begins with the Visualizer sending a subscription request to the Monitor, who acknowledges it with a reply. After

this initialization, the Monitor will keep sending data to the Visualizer whenever an update is received from the Power Manager. This will loop until an unsubscription signal is sent.

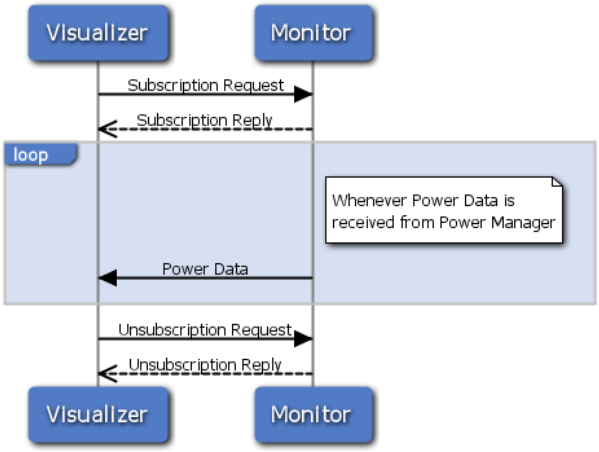

Figure 3.1: Subscription Protocol

Power Manager Monitor, Logging

Whenever the Power Manager has an update in its status, it will *log* this to the Monitor if it exists. So this protocol is very rudimentary and consists only of a check if the Monitor process can be found, and a simple data signal. This is illustrated in Figure 3.2.

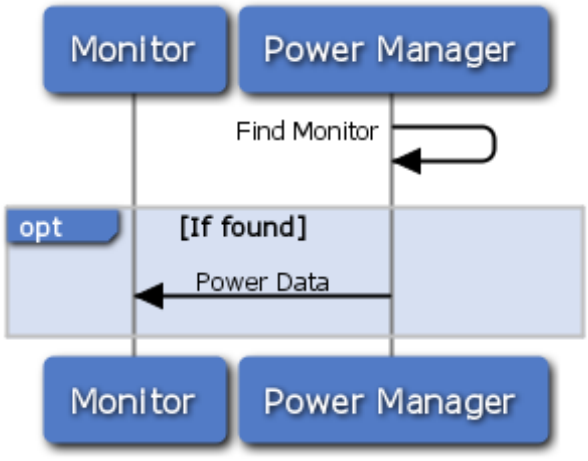

Figure 3.2: Logging Protocol

Visualizer Power Manager, Commands

The Visualizer can send two types of commands to the Power Manager: set a new required lifetime for the battery on the monitored node to last and set the power level of the simulated battery. There is no acknowledgement reply in Figure 3.3 since LINX is a reliable protocol. The same reasoning can be applied to the communication regarding setting the power level of the simulated battery.

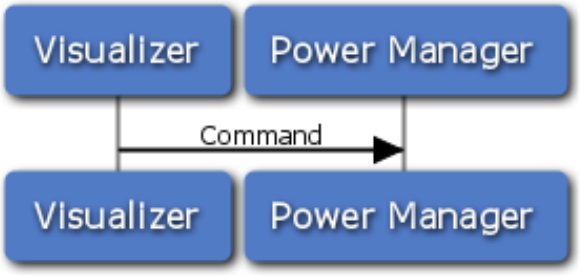

Figure 3.3: Commands Protocol

3.1.2 Firepad Central

In the Firepad – Central scenario, the only communication is the messages sent between the Firepad and the Central.

Firepad - Central, Register

Whenever a Firepad is started, it searches for the Central and registers itself. After receiving an acknowledgement, it starts sending information about its current condition. During these updates, the Firepad also listens for messages from the Central that request a new lifetime. This is depicted in Figure 3.4.

3.2 The Monitor

The monitor module will consist of three parts as shown in Figure 3.5. The parts will be implemented as two different processes and a minimal API. These parts are the *publisher*, the *terminal* and the *log API*. The reason for this division of processes is to follow the SRP (Single Responsiblity Principle, see Section 2.2). The two responsibility principles divided here are:

- Collecting power data (**REQ1-1**)
- Send power data to Visualizer (**REQ1-2**)

The API is designed to hide the inner workings of the Monitor from the Power Manager. In this way, the implementation of the Power Manager is little affected by the progress of the Monitor.

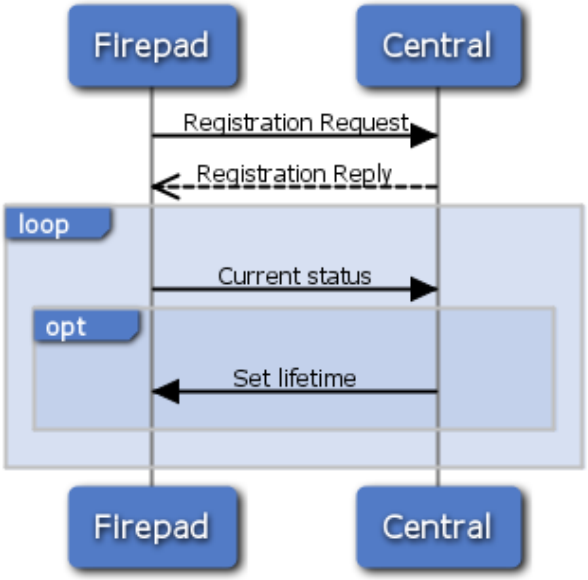

Figure 3.4: Registration Protocol

3.2.1 Publisher

The publisher is a process which main responsibility is to forward logged data, from a publisher, to a subscriber. It has two distinct states as shown in Figure 3.6.

The Publisher starts in the Waiting state where it waits for a subscriber to request a data subscription (which would be the Visualizer). When a subscriber has applied, the publisher goes to the Publishing state and waits for data to be logged (from a Power Manager).

From this point and on, whenever data is logged it will be forwarded through the publisher to the subscriber. The reason for this middleman is that data can be sent from the Power Manager on the board to the Visualizer, while the complexity in establishing connections can be moved from the Power Manager to the Publisher.

The subscriber can cancel this subscription of data by sending a special unsubscription signal to the publisher process. When such a signal is received, the publisher process will restart and wait until a new subscriber requests a subscription.

Observe that while in the Waiting state, any power data that might be sent to the publisher will be thrown away. This design choice was motivated due to the interest lies in live data, and therefore buffering it is of no concern.

3.2.2 Terminal

The terminal is a simple interface for a user to edit configurations of the Monitor application (for example for debugging). Nearly all user interaction with the Monitor will be done via Command messages sent from the Visualizer.

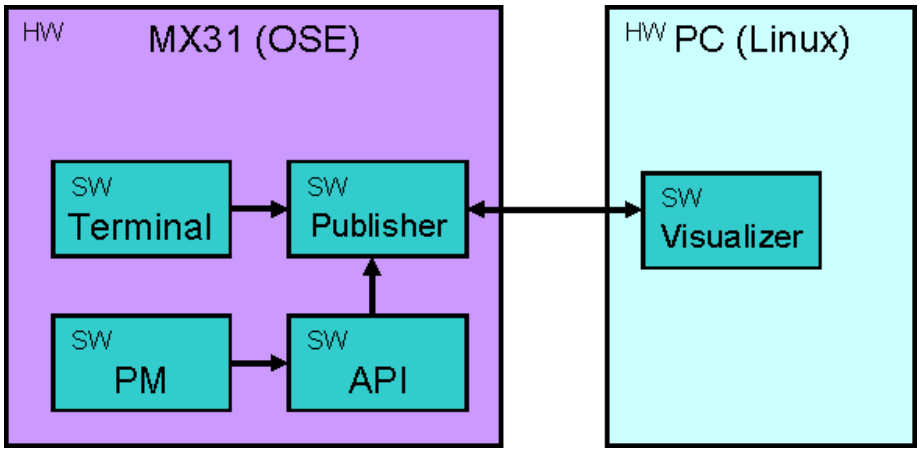

Figure 3.5: Design of the Monitor Module - Showing interactions with the PM (Power Manager)

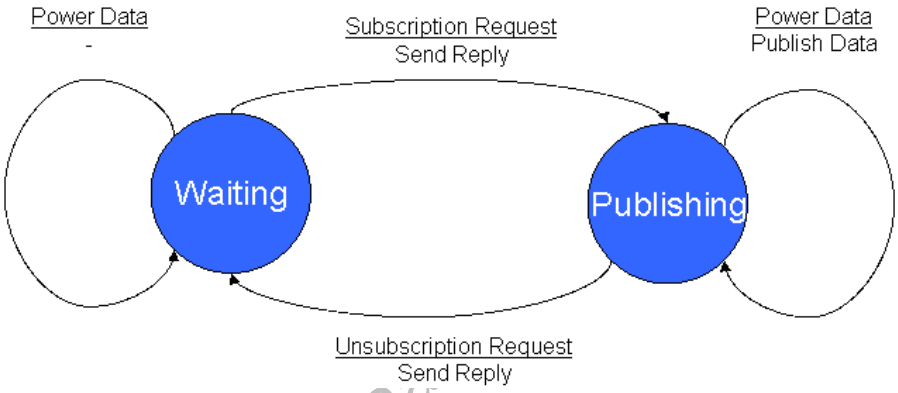

Figure 3.6: State Diagram of the Publisher

The most important feature of the terminal is the possibility to start a mimicking process that will behave like a Power Manager. This is useful to demonstrate the application without an implemented Power Manager.

3.2.3 API

The minimalistic API consists of a single function to log Power Data. It is used by the Power Manager to log data. This means that it will be sent to a publisher, if one is running. If there is no publisher running, nothing happens; this lets the Power Manager to not depend on the Monitor.

The design reduces the coupling between the Monitor application and the Power Manager. This means that from the perspective of the Power Manager, very little is known about the Monitor. This means that the Monitor part can easily be transformed, enhanced or completely rewritten without the need of change in the Power Manager's code.

3.3 The Visualizer

The Visualizer process is responsible of visualizing the data received from the Monitor. The Visualizer consists of three modules; these are the Main module, Signal module and the Animation module. The Visualizer will present the individual power consumption of up to three different PMCs (Power Manageable Components) of the monitored computer system. If the data of more than three PMCs are received, three components will be picked arbitrarily and visualized. This design choice has been made since during this thesis, there existed no more than three implemented PMCs.

3.3.1 Intra-Application Structure

Figure 3.7 shows the general outline of how the modules in the Visualizer interact with each other, the Power Manager and the Monitor, where the latter two both resides on the node monitored.

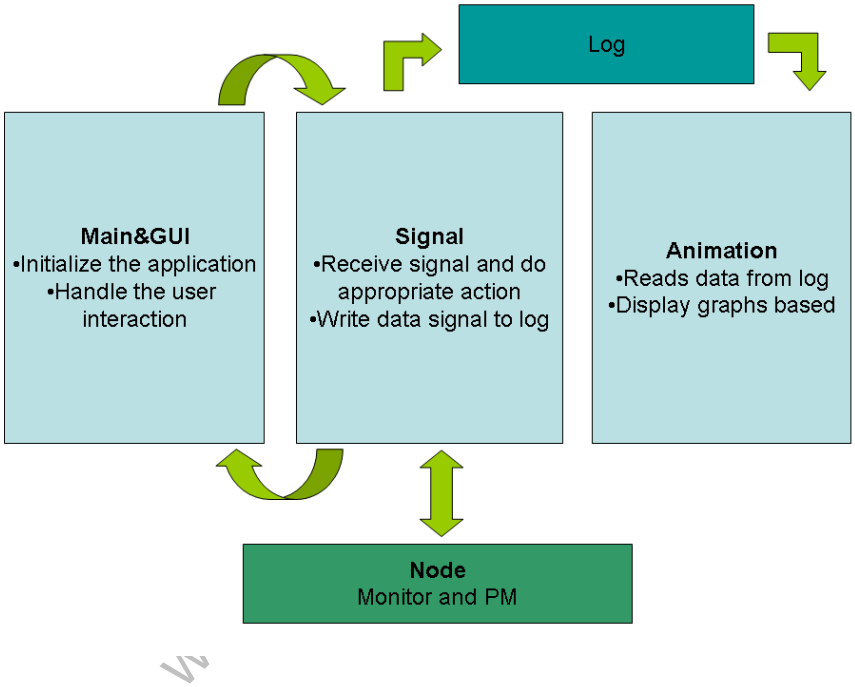

Figure 3.7: Structure of the Visualizer Module

The Main thread will be responsible of handling the user interaction with the GUI by responding to events triggered by the GUI and delegate the responsibility of the sending appropriate signals to the Signal module. The Signal module will in its turn send the corresponding signal to the node.

The Signal module will receive every signal of Power Data from the Monitor and write it to the log. The animation threads will later read from the log and visualize the data with diagrams. Also, the Signal thread may have to trigger a change in the GUI (for example disabling a button) and therefore it needs to be able send requests to the Main module.

The interaction of the Animation module, the log, the Signal Module and the GUI is inspired by the Model-View-Controller pattern. The MVC pattern is used to decouple the handling of input, processing of data and displaying of output by separating them into a Model (processing of data), a View (displaying of output) and a Controller (handling of input)[1].

In this application the Model is the log that stores the data, the View is the diagrams and the Controller is the interactive GUI that accepts input from a user. By using the MVC pattern, the application is divided into seperate modules. The complexity of each module is then reduced and the possibilities to change a single part without affecting the others (for example using another way of presenting the data) are increased. The design of the modules also strives to adhere to the SRP (Single Responsiblity Principle, see Section 2.2).

These patterns made a clear distinction between what modules that were needed and furthermore this design attempted to make the modules as much as possible decoupled from each other.

3.3.2 Inter-Application Structure

Figure 3.8 illustrates the sequence of signals sent between the Visualizers modules and the Monitor when the user triggers a subscription request from the GUI:

1. The Main module sends the Signal module a message describing that it should send a subscription request to the Monitor.
2. The Signal module sends a subscription request.
3. The Monitor replies and begins sending Power Data.
4. The Monitor continues to send Power Data until the user unsubscribe from the Monitor.
5. When the user requests to unsubscribe, an unsubscription request is sent.
6. The Monitor replies and confirms the unsubscription.

Events in these modules will happen asynchronously of each other and thus parts of the different modules shall run in different threads.

There shall be five threads in total as shown in Figure 3.9, one in the Main module to handle the GUI, one in the Signal module to listen for signals and three in the Animation module to draw the three different graphs. Another possibility would be to implement the threads as processes, however threads can easily share memory and they are more lightweight than processes.

3.3.3 Main Module

The Main thread will run as a part of the Main module, it will run the necessary initialization functions and after that it will only respond to events that occur in the user interface. It shall be able to propagate commands from the user to the Signal module.

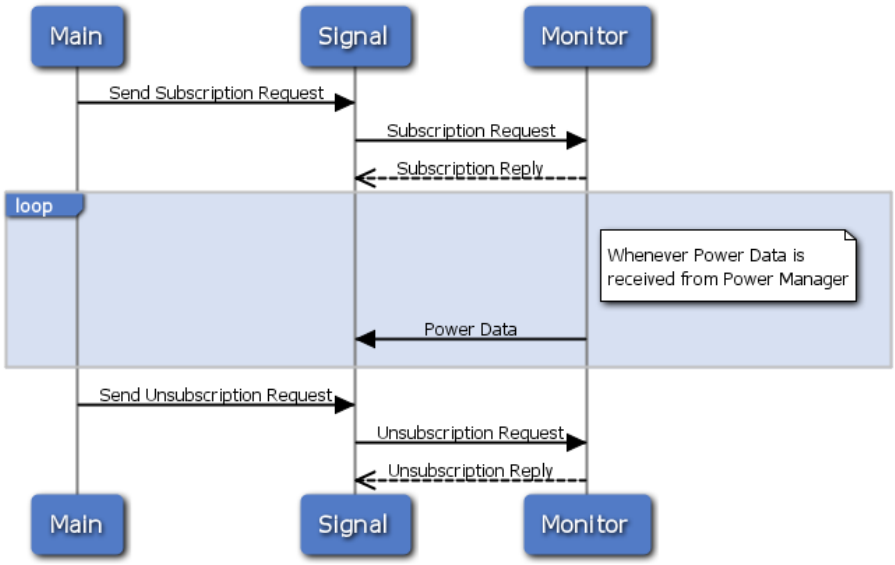

Figure 3.8: Inter-Application Communication

3.3.4 Signal Module

The Signal thread will be a part of the Signal module and it will be signal-driven. When a signal arrives, the corresponding action for that signal will be taken. This process will be repeatedly indefinitely. This thread will fulfill a part of **REQ1-2** (see Section 2.2.1) since if the Monitor shall be able to send data to the Visualizer, the Visualizer must consequently be able to receive data from the monitor and thus fulfill a part of the requirement.

The Signal Module shall also be able to send commands to the Power Manager, to set either lifetime or battery level of the node. This will be initiated by a user and the GUI module will call the Signal module which will contact the Power Manager. This is a part of the requirement **REQ1-5**(see Section 2.2.1).

There will be three types of signals that are received or sent: subscription or unsubscription request, set new battery level/lifetime and a data signal containing the Power Data collected at a node.

A subscription request will ask the Monitor to start sending the data it supervises. This fulfills **REQ1-4**(see Section 2.2.1 by establishing the subscription. An unsubscription request will request the Monitor to stop sending data.

The set lifetime signal will be sent to the Power Manager which should instruct the device to last the required time. Any value given as input to the Visualizer will be propagated to the Power Manager, even if it would be an unreasonable amount of time. The result of the request depends on the implementation of the Power Manager.

The battery is simulated, therefore a signal to set a new battery level can exist, and this will tell the simulator to adjust to the given power. The reason for a simulated battery is that during the time of the project, the used hardware did not have an actual battery.

The data signal contains Power Data collected at the monitored system.

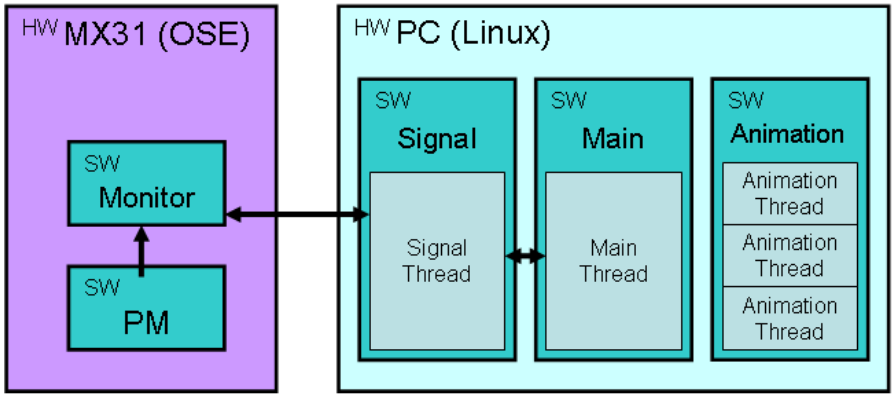

Figure 3.9: Threads in the Visualizer Module

When the signal handler receives a data message it writes this to the log which at some point will be read by the animation thread. In this prototype the problem of having a max size on the log is not consider. It will be dynamically growing, and can therefore handle very large amounts of data. But of course the memory can be full, and the program will then crash. This problem is ignored to avoid unnecessary complexity in this prototype.

3.3.5 Animation Module

An animation thread is responsible to draw a graph. There will be three of these threads since the GUI will display three different graphs. An animation thread will poll a log a number of times every second to get the values it needs to display. The third requirement **REQ1-3** (see Section 2.2.1) is fulfilled by the animation threads.

The animation thread will start when the Signal module receives a subscription request message and run until it receives a request to stop message

3.4 The Firepad

The Firepad will be a simple prototype demonstrating some of the possibilities of power management. Its structure is very simple; it contains a single OSE process. The behavior of the process is depicted in the sequence diagram in figure 3.10.

It is stated in requirement **REQ2-2**(see Section 2.3.1) that the Firepad shall find the Central automatically, so it starts by searching for the Central process until it is found. This will be done using LINXs hunt call2.1.2. A real Firepad would of course start helping the user right away, for example by showing temperature, position, etc. But in this prototype there is no interest in running this process unless a Central is there to observe (since no firefighter is actually using this application).

When the Central has been found, the Firepad sends a registration request, to apply for observation. This is done so that some vital but static information

(such as the firefighters name) can be sent to the central only once and need not be sent with every data message. Since the Firepad is the one locating the Central, it is obviously more logical to let the Firepad register than the other way around.

If a reply would not be received, the Firepad goes back to the searching state (this could happen if the Central just crashed). However, if a reply is received, data is read from the reply (such as information about surrounding environment) and stored accordingly. After this, it reaches the "Do Actions" state which in this prototype scenario equals simulating movement and sensors. In a real application, it would also check for user input and display output in various forms.

The next thing to be done is to check for any incoming messages, in this prototype, the only interesting message is the request of a new lifetime (**REQ2-4**, see Section2.3.1. If such a request is received, it has to be propagated to the Power Manager. This approach has been taken since it is desired that the Central application has as little knowledge of the Power Manager as possible (see Section 2.3).

After the actions have been completed, the status of the Firepad might have been updated (e.g. new position) and therefore it needs to send a data message to the Central with the new information (as required by **REQ2-3**)(see Section 2.3.1).

Another approach would have been to create a separate process that would send an update at a predefined interval (e.g. once every second). The multi-thread approach will not be taken since it is unnecessary to send data to the Central if the current condition has not been changed.

When the data have been sent, the Firepad will check whether it has lost connection with the Central, in that case it will go back to finding it; otherwise it will loop.

3.5 The Central

This component of the Firefighter scenario, the Central, is similar to the Visualizer application in the computer system monitor scenario (see Section 3.3. However the main difference is that the firefighter Central will have to receive data from several different nodes. Furthermore, Firepads needs to register at the Central instead of the Central registering the Firepads.

The fireman central will consist of three modules: the GUI module, the Drawing module and the Signal module.

3.5.1 General Outline of the Central

This application will consist of two threads one for the GUI module and one for the Signal module. The signal thread will wait for signals; if it receives a register request it will register the Firepad. If it receives an update signal from a Firepad it shall redraw the map with the new Firepad information.

Updating the displayed information about the firefighters will be driven by how often the Firepads sends new update information about their current condition. The thread that receives new information will interact with the drawing

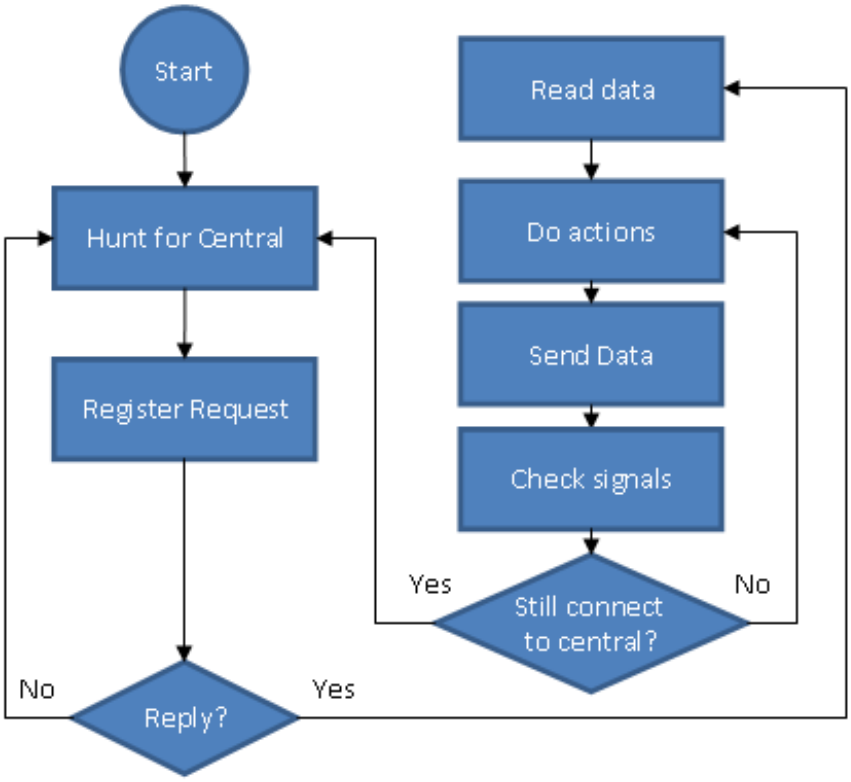

Figure 3.10: Flow of the Firepad Program

module to display the updates every time it receives an update from a registered Firepad. Figure 3.11 illustrates this sequence.

The reason for choosing an event-driven update of the drawing module (instead of a timer-based) is to reduce complexity. With this design, there is no need to implement an extra thread that keeps track of time. Also, there will be no unnecessary redrawing, the only problem would be if there became to much redrawing. But in this prototype, the maximum number of Firepads simultaneously connected will not be too many. If this would become a problem, the timer-based approach could be taken instead.

3.5.2 GUI Module

This module shall create the GUI and afterwards it should handle the events that occur in the GUI. This includes taking care of selecting a specific Firepad (to be monitored and commanded).

3.5.3 Signal Module

The signal module will handle the connection and communication with the Firepads. It will register Firepads when they report that they are up and run-

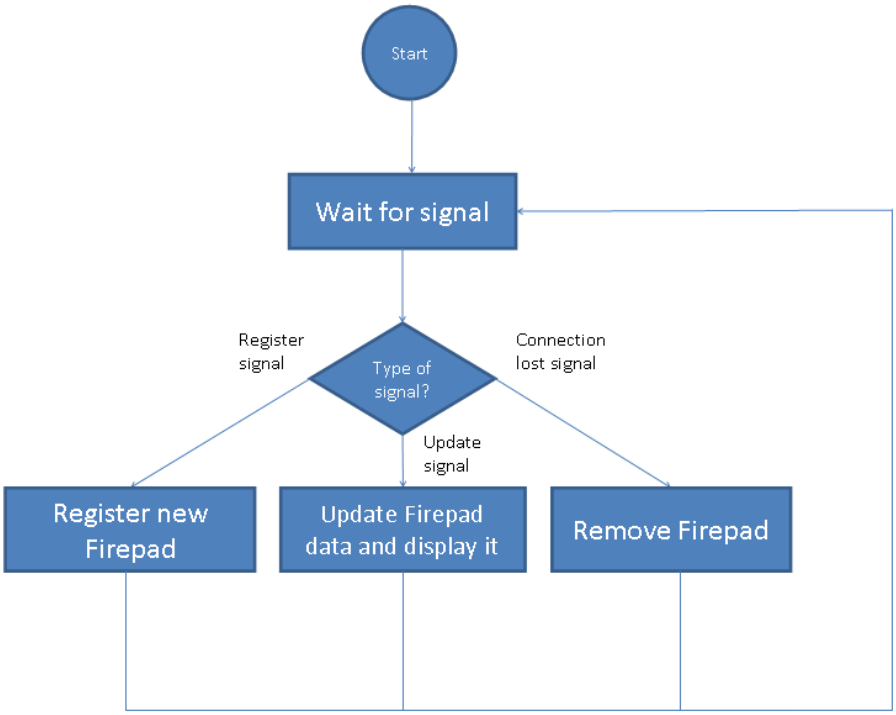

Figure 3.11: Flow of the Firefighter Central Module

ning which will cover requirement **REQ2-2**(see Section 2.3.1). Requirement **REQ2-3**(see Section 2.3.1) states that when a Firepad sends data about its current condition the Signal module shall receive the information and act accordingly. It shall also fulfill **REQ2-4**(see Section 2.3.1), thus it shall be able to send a new lifetime to a specific Firepad.

3.5.4 Drawing Module

The drawing module will have contain functions for drawing the overview of a map with the Firepads position on it as well as more specific data of a selected Firepad. Clearly **REQ2-1**(see Section 2.3.1) shall be fulfilled by this module.

3.6 Summary of Fulfilled Requirements

All the requirements have been met with this design. And therefore if this is implemented, the resulting applications will demonstrate the effects of power management.

Requirement	Fulfilled	Section
REQ1-1	Yes	Section 3.2
REQ1-2	Yes	Section 3.2
REQ1-3	Yes	Section 3.3.5
REQ1-4	Yes	Section 3.3.4
REQ1-5	Yes	Section 3.3.4
REQ2-1	Yes	Section 3.5.4
REQ2-2	Yes	Section 3.4
REQ2-3	Yes	Section 3.4
REQ2-4	Yes	Section 3.5.3

Table 3.2: Summary of Fulfilled Requirements

Chapter 4

The Implementation of the Design

The result of the work consists of the two applications designed. During the implementation of these several problems arise and were dealt with. One early decision taken was to use LINX for communication. Here follows first an introduction to LINX, and afterwards the implementation details of each application.

4.1 The Monitor

Implementing the Monitor was straight-forward by following the design (see Section 3.2). The application consists of two processes, one called the Publisher and another one called the Terminal (representing the corresponding modules in the design).

The Publisher (see Section 3.2.1) is in practice a big loop with the two states from the design, using LINX signals to do all communication (both with the Power Manager as well as the Visualizer).

The Monitor is designed such that there can only be one subscriber at a time. There is nothing stated at what should happen if a new subscription request is received, when there already is a subscriber. The decision in the implementation became to remove the old subscriber in favor of the new one. This was the logical choice because if the opposite would be chosen (to keep the current), a subscriber could "lock up" the Monitor and never let it go.

The Terminal (see Section 3.2.2) is also a loop, printing to and reading from standard input (using standard C I/O-functions). Lastly the API was very small and contained no surprises.

4.2 The Visualizer

The GUI was implemented with the Glade[1] and GTK+[2]. GTK+ is a graphical user interface toolkit for creating GUIs for the X Window System.

[1] http://glade.gnome.org/
[2] http://www.gtk.org/

A decision was made to use GTK+ as the library for creating the user interface in the Visualizer since attempts with Ruby[3] and its libraries resulted in unsatisfactory performance.

Glade is a graphical user interface designer and one of the reasons for choosing Glade is the ease of creating a GUI with pointing and clicking, thus it is possible to get immediate feedback of how the design looks.

The communication with the Monitor or the Power Manager on the Signal module was implemented with the help of LINX for the communication. The Animation module used the PLplot[4] library to plot the graphs in real-time. PLPlot was used for the reason that the library is able to plot graphs in real-time which is the main purpose of this task.

There were a few minor discrepancies between the implementation and the design in the Visualizer component.

For example the diagram below illustrates the process of beginning subscribing data from a Monitor. Here the Main Thread would sends a subscription request to the Monitor process, however when using LINX it is not possible to send a message to a process with the name only. Thus a hunt call needs to be made and the exchange of signals differs from the design and became like shown in Figure 4.1. The Animations threads are not dynamically started when a Subscription Reply is received because of problem that was encountered.

When starting the animation threads dynamically when a Subscription Reply was received, the signal thread would supposed to receive a subset of all LINX messages, yet it seemed like it would receive all possible messages thus creating a race condition of who would receive the stop animating request between the receive thread and the animation thread. Therefore the animation threads will be started when the Visualizer component is started.

The sequence diagram illustrates the successful case from subscribing to unsubscribing from the Visualizers perspective.

The first event that is triggered is that a user presses a subscribe button, the corresponding callback function for that button is called and a hunt signal is dispatched to get the location of the Monitor process. The hunt signal reply indicates that a Monitor process is found by LINX. A subscription request is then sent to the demo process and a reply acknowledgment is received. The Animations threads reads from a log at a specific interval, the first time nothing is written in the Log. The Signal thread writes data to the Log when it receives a Data signal which contains the data to be written. When the user presses the unsubscribe button, a similar sequence to the subscribe button is executed, however this time the address of the Monitor process is known therefore a unsubscription signal can be sent to it directly.

One of the larger obstacles of the Visualizer was creating the GUI. In the beginning the graphical user interface was made in Ruby since it is an easy scripting language and therefore a mock-up of a GUI could be rapidly created. In addition the animation could be easily implemented since it exist libraries for drawing graphs to visualize the data. For creating the graphical user interface, three different GUI toolkits: Shoes, Tk and Ruby-GNOME2 were used.

The major problem was that the Scruffy library used, a library to generate vector based graphs did not seem to be developed to render graphs in real-

[3] http://www.ruby-lang.org/en/
[4] http://plplot.sourceforge.net/

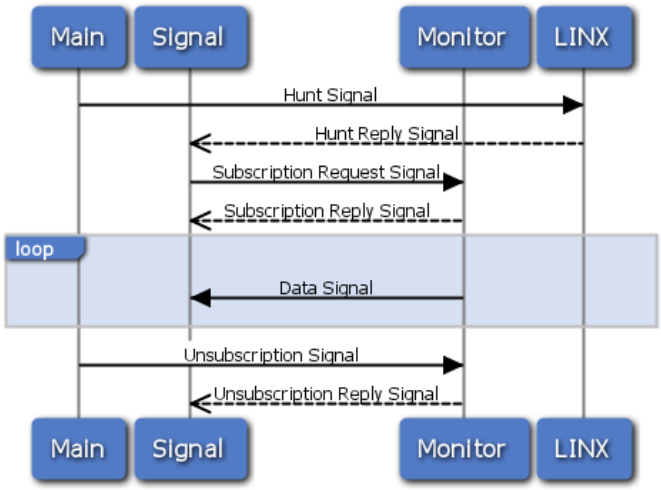

Figure 4.1: Actual Signal Sending

time. The rendering and displaying the images was only possible a few times per second, this resulted in three or four graphs could only be updated once a second if the images were small enough. The larger the graphs were the longer the process took to render them consequently the process became so slow that the program could not handle it and soon it would crash.

Functions in Shoes were made which would draw directly on a canvas but the same setback occurred again namely the animation together with the communication module resulted in that the program could not handle the drawing and data parsing under those conditions and crashed.

With these problems in mind a decision was made to use another language since Ruby and its libraries were to slow for the application requirements. The decision was to use Glade with GTK+ a programming language independent GUI concept. The consequence is a flexible system, without risk to re-write the graphical part of the software in case a change to another programming language.

4.3 The Firepad

The Firepad consists of a single process. The behavior of the Firepad is conforming to the sequence diagram defined in section the design. Communication is also here done solely by LINX.

If a request of lifetime is received, it is propagated to the Power Manager by hunting for it, and then sending the message. This would of course be a problem if there is no Power Manager running, but this application assumes its existence.

4.4 The Central

The implementation of the Central application was very similar to the implementation of the Visualizer application. There are however some differences, there are two threads in the Central application. One thread is listening for signals from the Firepads and the other one is handling the GUI. The animation is this time driven by how often the Firepads new information.

The animation was done with the help of the Cairo[5] library which is a library for creating vector graphics and one of its backends is the X Window System. We used it since vector graphics looks good on a display and moreover it seemed easy to create simple shapes and figures since vector graphics deals with geometrical objects such as curves and lines.

The communication for the Signal module is again implemented with the Enea Linx for Linux. GTK+ and Glade was also used in this application to create the GUI.

The Central can receive three types of signals: a register request signal, updated data signal and a lost connection signal.

When the Central receives a register request signal it will allocate space for a Firepad, map the name it sent to an ID and it will with the help of LINX attach to the Firepad. Attaching to a Firepad means that if the connection is lost, a signal will be sent to the Central from a LINX process and thus the allocated space for the Firepad can be removed.

When the Central receives an update signal from a Firepad it will update the data of the corresponding Firepad data in the Central. It will subsequently draw redraw the representation of every Firepad.

[5] http://cairographics.org/

Chapter 5

Conclusion

After the implementation of the applications, there are several questions to be answered. The most important one is if the resulting software fulfils the requirements. Following is a review of the requirements presented in the architecture sketch, and afterwards a discussion about the question asked in the introduction.

5.1 Review of Requirements

By considering the requirements one by one, assurance can be made that the software fulfils the functionality required.

REQ1-1: The Monitor shall survey the Power Manager and collect Power Data.

This is fulfilled by the Monitor receiving LINX signals from the Power Manager. However, no real surveillance of the Power Manager is done (all communication is one-way, from Power Manager to the Monitor). This could be a problem if the user wishes to know whether the Power Manager is alive or not. But in this implementation, only the data is monitored since that is the primary interest.

REQ1-2: The Monitor shall be able to send power data to the Visualizer.

By using inspiration from the subscriber/publisher pattern, the Monitor can get a subscriber (Visualizer) and send data to it, in the form of LINX signals, as long as it is requested.

REQ1-3: The Visualizer shall display the power data received from the monitor.

This requirement is fulfilled by receiving LINX in a signal listening thread which writes the received data to a log. The log is polled from an animation thread which will plot a graph in real-time to visualize the power management effects.

REQ1-4: The Visualizer shall be able to establish a connection the Monitor.

This is fulfilled by the Visualizer hunting for a process named Monitor. The Visualizer can then send a subscription request to the Monitor and thus a connection has been initiated.

REQ1-5: The Visualizer shall be able to send commands to the monitored system.

This requirement is fulfilled by hunting for the Power Manager with the help of LINX and once the Power Manager has been located, a command can be sent to it.

REQ2-1: The Central shall display current condition of the connected Firepads.

The Central receives messages from the Firepads, and when an update message is received the Central will with the Cairo library redraw the representation of all Firepads. Clicking on a specific Firepad in the Central extra information about that Firepad will displayed.

REQ2-2: A Firepad shall be able to automatically find the Central and connect to it.

By using a hunt call, the Firepad searches for a specifically named process on a specific host (i.e. a host with a specific name) in this implementation. When found, a registration request (as a LINX signal) can be sent. By searching for a specific process on a specific host can of course be a problem in a more general scenario, but in this is prototype it is acceptable.

REQ2-3: A Firepad shall be able to send data about its current condition to the Central.

With the use of LINX signals, the data can be sent to the Central after a registration request/reply has been exchanged. The data is in this case generated in the Firepad process and then sent. In a real application, this data would instead be collected from sensors.

REQ2-4: The Central shall be able to set required battery lifetime of a Firepad.

This requirement is fulfilled by selecting a Firepad in the Central and if clicking on sends. The Central will send a signal including the required battery lifetime to the Firepad which will contact the Power Manager to set the lifetime.

5.2 Power Management on GEODES

Power management can definitely be implemented on a distributed embedded system, the Firefighter scenario is a working prototype demonstrating this, under the assumption that the Power Manager and the Power Manageable Components can be implemented.

In the Firepad, the feature of setting a requested lifetime demonstrates that Power Management can be hidden below a layer of abstraction. This lets the application developers to utilize the features of Power Management without extensive knowledge in the area.

It has also been shown that the Power Management is possible to be deployed on a heterogeneous distributed system. Since in the firefighter use case, two different kinds of system has participated in the communication. Also, no assumptions on the underlying hardware or software has been taken from the Centrals view, only that the specified communication protocol has been implemented.

5.3 Developers' Toolkit

During the course of this thesis an investigation about what should be included a Developers' toolkit from the perspective of an application developer. This Developers' toolkit would include among other things an API to be used by a developer of applications that could benefit from the use of power management.

A developers' toolkit should have a well defined interface to an application that is taking care of the power management features, since this seems to be the easiest way to control the power management. For example setting a required lifetime could be done by calling a function with the desired value.

One problem in the firefighter scenario is that the fire brigade leader cannot examine a firefighter's device to see how long it is required to survive (the leader can only set this value, not read it). This is problematic, since it becomes impossible for the leader to get an overview of the current situation. Therefore, it would be a very good function to have in a developer's toolkit.

Also, a function to shutdown all but the necessary functions for a system to run is also a good feature e.g. a mobile phone may shutdown everything except what is needed to make an emergency call. This could be generalized such that a single (or a few) components can be prioritized, such that they should be kept at an higher performance level than the rest of the system.

5.4 Future Work

Since this project lasted only ten weeks, a lot of ideas were either implemented hastily or not at all. This has led to the existence of some implementation details of lesser quality; this is also a downside of agile software development. Some of these rough edges are brought into the light here to explain how things could have been (or can be done) differently in the future to enhance the application.

5.4.1 The Monitor

The final version of the Monitor does not implement the publisher/subscriber pattern completely, but only a part of it. The lacking detail is the fact that the Monitor only allows one subscriber. In this version it's not a problem, since the requirements (**REQ1-2**, see Section 2.2.1) only asks for the capability of sending data to the Visualizer. It could definitely be of value to have several Visualizers monitoring the same node (for example if the Visualizers are in different places geographically).

A very important flaw in this experiment is that the battery is simulated. This leads to a problem since that means any examination of the effects of power management with respect to power drainage, is ultimately based on the quality of the simulation. In this thesis, such observations has not been done, it has only examined the effects of state-based situations.

By adding the functionality to read the remaining power from a real battery would fix this flaw and give new possibilities of observing the effects of power management.

5.4.2 The Visualizer

In terms of functionality there is no major component that seems necessary to add to the Visualizer.

However, there are some improvements that could be made to the Visualizer: making the graphs larger, more fashionable, showing them in a separate window or add a resizable feature.

5.4.3 The Firepad

The main thing to be added to the Firepad application is some kind of output that demonstrates the effects of the power management on the local hardware. A prototype OpenGL application was developed (that would display temperature) which would be affected by the local Power Manager. This was not used in the final prototype since the OpenGL wasn't implemented on the used operating system.

5.4.4 The Central

A problem of mostly theoretical nature in the Firefighter Central application is an error due to excess amount of data received. The data received from the Firepads drives the animation rate. Thus if the Firepads are too many Firepads on a map, they could send too much data for the central to process and as a result may crash. An improvement and solution that could be implemented is to buffer and filter the data received by the Firepads.

A more extensive expansion of the Central could be to implement an algorithm that controls the firefighters. This would require an elaborate use case (by adding more parameters to the scenario) to give a computer controlled Central more input to work on. A possible addition could be requirements of a minimum number of firefighters at the scene at all times. This would require the Central to prolong firefighters required lifetime if new firefighters are not coming to the emergency scene in time.

Tools

Cairo

Vector graphics library
http://cairographics.org/

Glade

GUI builder
http://glade.gnome.org

GTK+

GUI library
http://www.gtk.org

LaTeX

Typesetting library
http://www.latex-project.org/

LINX

Inter-Process communications library
http://linx.sourceforge.net/

PLPlot

Graph plotting library
http://plplot.sourceforge.net

WebSequenceDiagrams

Sequence diagram tool
http://www.websequencediagrams.com/

References

[1] Frank Buschmann, Kevlin Henney, and Douglas C. Schmidt. *Pattern-Oriented Software Architecture Volume 4: A Pattern Language for Distributed Computing*. John Wiley and Sons, 2007.

[2] Simon Eriksson. *System Analysis of Energy-Constrained Quality of Service and Power Management Techniques*. Royal Insitute of Technology, Department of Machine Design, 2010.

[3] Eric T Freeman, Elisabeth Robson, Bert Bates, and Kathy Sierra. *Head First Design Patterns*. O'Reilly Media, 2004.

[4] Robert C Martin. *Agile Software Development, Principles, Patterns, and Practices*. Prentice Hall, 2002.

Appendix A

Manual for Visualizer

To visualize the data monitored on a board, a subscription has to be established to that node. To subscribe to a node (Figure A.1):

1. Choose a node to subscribe to in the combobox
2. Click on the subscribe button

Illustration of subscribing to a node:

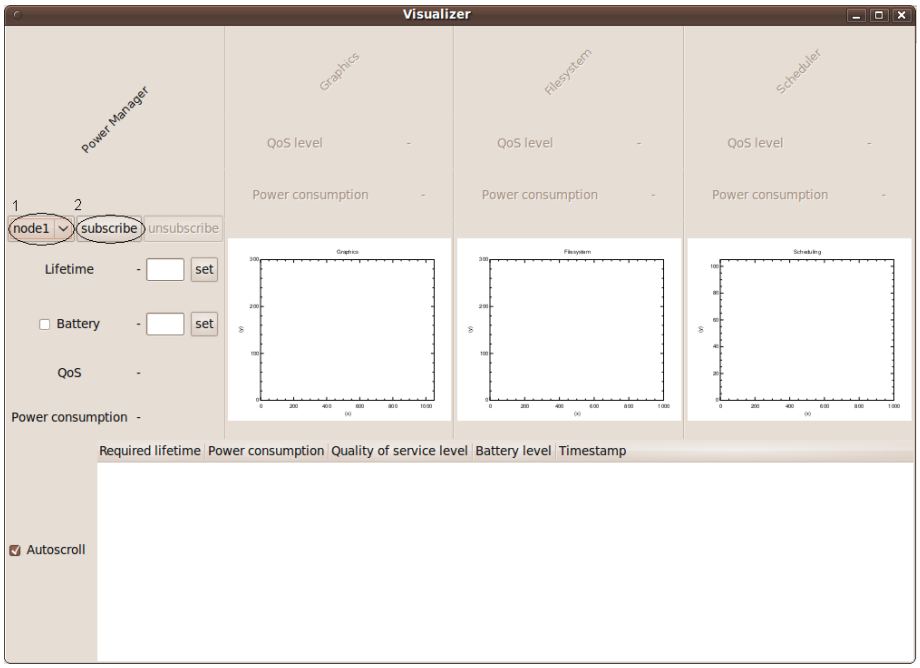

Figure A.1: Visualizer in Action

To stop receive data from a node unsubscribe from it (Figure A.1):

3 Click on the unsubscribe button and the node subscribe to will be unsubscribe.

To set a new required lifetime of a node that is subscribed to, you need to send it a command to do so. To set required lifetime (Figure A.2):

4 a) Put a value in the entry box

4 b) Click on the set buttons

To set new battery lifetime:

5 Click on the battery checkbox to get an infinite amount of power.

Or to set a specific battery level:

6 a) Put a value in the entry box

6 b) Click on the set buttons

Additional feature:

7 To auto scroll among the received messages check the Autoscroll box, if you want to look further at a message uncheck the auto scroll and go to that message.

Figure A.2: Subscribing a node

Appendix B

Manual for Firefighter Central

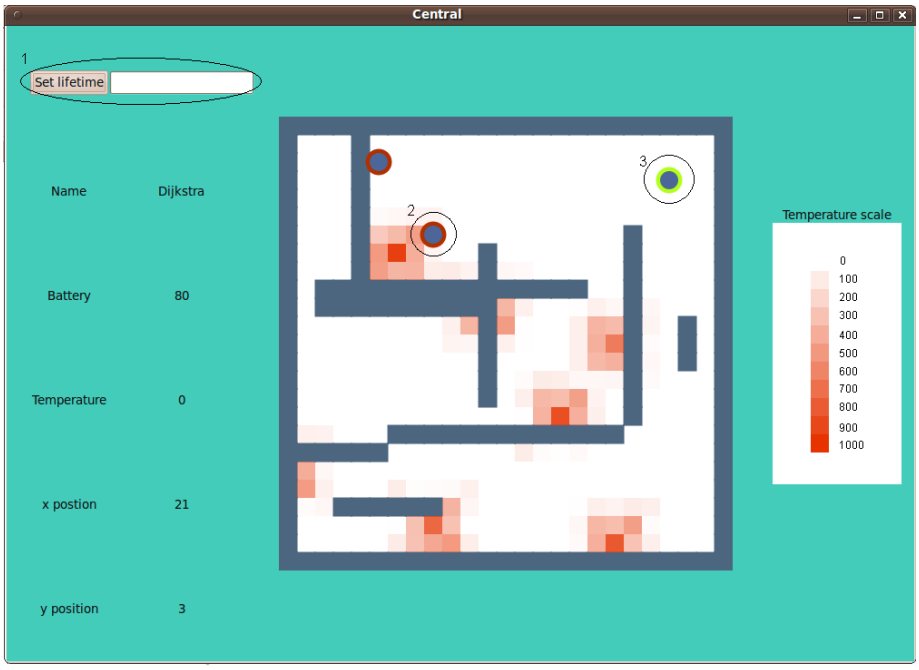

Figure B.1: Firefighter Central Screenshot

Firepad represented by a blue dot with a ring around it. A green ring (3) represents the Firepad selected and its data is show to the left of the map. A red ring (2) around the Firepad represents a Firepad that is not selected and thus his data is not shown anywhere except the graphical representation of his current coordinates.

The red rectangles represent a temperature and which scale is shown to the right.

To set a new required lifetime of the selected Firepad:

1 a) Write a number in the entry box

1 b) Click on the Set lifetime button