Wei Li

Event-Driven Resource Management on Mobile Devices

School of Electrical Engineering

Thesis submitted for examination for the degree of Master of Science in Technology.

Espoo 18.11.2011

Thesis supervisor:

Prof. Antti Ylä-Jääski

Thesis instructor:

M.Sc. (Tech.) Yu Xiao

Aalto University
School of Electrical Engineering

Aalto University
School of Electrical Engineering

ABSTRACT OF
MASTER'S THESIS

Author:	Wei Li		
Title:	Event-Driven Resource Management on Mobile Devices		
Date:	November 18, 2011	**Pages:**	78
Professorship:	data communications software	**Code:**	T-110
Supervisor:	Professor Antti Ylä-Jääski		
Instructor:	Yu Xiao M.Sc. (Tech.)		

Smartphones have become more powerful than ever that they can process complex tasks and handle heavy wireless data transmission. However, the increasing energy consumption caused by the increasing workload poses a challenge to the battery life, since the battery industry has not been able to develop as fast as mobile computing techniques. Hence, how to manage the resource consumption on mobile devices becomes an essential topic.

In this thesis we propose an event-driven framework for resource management on mobile devices. The idea is to use events to describe the changes in contexts and to control the consumption of resources based on events. Our framework adopts publish/subscribe mechanism, which allows applications and other software components such as power management software to subscribe to the events they are interested in.

We evaluate our framework with two power management applications: SNR-based transmission adaptation and traffic-aware power management of Wi-Fi network interface. In the first case we manipulate the Wi-Fi network interface based on SNR value predictions. In the second case we modify the Wi-Fi network interface and TCP settings based on the TCP burst predictions. Our test results have verified our target: with proper definition of the events rules, the resource is manipulated according to the adaptations and some energy is saved.

Keywords:	resource management, event-driven framework, publish/subscribe
Language:	English

Preface

This thesis is to accomplish my study in Aalto University as a master's student. When I entered the university in the year of 2007, it was still called Helsinki University of Technology. In the four years study since then. I really enjoyed the life here. I hope this thesis can be a good ending of my memorable study.

Firstly I would like to thank Professor Antti Ylä-Jääski, who gave me the chance to do this thesis. Computer science and engineering is not my major but my minor study. So I am grateful for Prof. Ylä-Jääski who trusted me and allowed me to do this subject. I also want to thank professor Risto Wichman from my major study, communications engineering. It was him who supported and gave me the freedom to choose the topic I wanted.

Of course I would also like to thank my instructor, Miss Yu Xiao. She gave me fully support, and helped me by pointing me the research direction, and designing some algorithms we used in the projects. She was always with patience when I was having some problems, sometimes even some stupid ones. And she endured me that I did this thesis a little bit longer than we expected because I had to work at the same time.

Last but not least, I want to thank my girlfriend, Miss Min Wu, who always believed in me, and encouraged me when I had problems. Not only did she support me in my life, but also inspired me in the work.

And there were also many other friends of mine who helped me. I couldn't finish it without them. I want to thank all of them because I really enjoyed the work with their help.

Espoo, November 18, 2011

Wei Li

Abbreviations and Acronyms

WNI	Wi-Fi Network Interface
SNR	Signal to Noise Ratio
CDF	Cumulative Distribution Function
TCP	Transmission Control Protocol
PSM	Power Saving Mode
WLAN	Wireless Local Area Network
SSID	Service Set IDentifier
HAL	hardware abstraction layer
IEEE	Institute of Electrical and Electronics Engineers
IP	Internet Protocol
IPC	Inter Processing Communication
XML	Extensible Markup Language
OS	Operating System
IPC	Inter Processing Communication
UML	Unified Modeling Language
UI	User Interface
GUI	Graphics User Interface
OOP	Object-Oriented Programming
XML	Extensible Markup Language
HAL	Hardware Abstraction Layer
ISO	International Organization for Standardization
WWW	World Wide Web

Contents

Chapter 1

Introduction

1.1 Motivation

Mobile devices have gained huge popularity in the past two decades. At the same time the system complexity of them has also increased exponentially. The first handheld cell phone was produced in 1973 in the United States of America. Since then, the technology has evolved with tremendous pace. By the year of 1993, the first Smartphone, IBM Simon, was launched by a joint venture between IBM and BellSouth. The functionality of IBM Simon was complex even by current standards: it had calendar, calculator, contact book, world clock, note pad, email client, and function to send and receive faxes. It even had a touchscreen for input. It also introduced another feature to the customers: the possibility to combine a mobile phone with a PDA (Personal Digital Assistant). [1] But because of the expensive price, the unfriendly user experience, and limitations on such fields with network access and computing power, it did not become a huge success at that time.

Along with phone evolving was the mobile access technology. The first second generation (2G) network was launched in 1991 in Finland, which was using the GSM standard. The 2G system became digital and hence more digital integrated circuit (IC) was used. As a result, 2G system provided better phone quality, smaller size, as well as cheaper price. The next generation, 3G, advanced by providing higher data connection speed, which enables the user to access to Internet anytime and anywhere. As a result, nowadays the smart phones are small in size, have high data transmission speed as well as powerful computation ability. A modern smart phone typically has a powerful CPU

[1] The term PDA was firstly created in 1992

(nowadays the frequency of CPU can be over 1GHz), a big screen with high resolution, a large memory capacity (at least 512 MB RAM), mass storage space (as high as 32 GB), multiple connectivity including 2G/3G, Wi-Fi, Bluetooth, USB, as well as GPS (Global Positioning System) integrated. These changes can be seen when comparing two produces, such as Nokia 9210 communicator (released in 2000) and Nokia N900 (released in 2009), as illustrated in Figure 1.1. We can see that within 10 years, most of the system resources have become at least 20 times more powerful.

Figure 1.1: Comparison between two smartphones released in 10 years

The software on mobile phones has also been going through rapid change. There are currently many mobile operating systems, including iOS, Android, Meego (formerly Maemo), Symbian and Blackberry OS (these systems all have the hardware requirements mentioned above). The available software range varies from application to system middleware, and the number of them has been increasing incredibly. For example, Apple launched its online software store for iPhone at July 10, 2008, and by July 2011 there were more than 500,000 applications online (and these are not even all of applications on iOS, because not all SW are authorized to be published on the App Store).

Another breakthrough at that time was the rapid growing popularity the Internet. As the data transmission increased, people started to think about the possibility of using the Internet over the mobile phone. After all, it would be tempting to access the Internet whenever they want and wherever they are. To satisfy such requirement, data transmission speed over mobile networks has been increased dramatically. For example, in HSPA, which

stands for High Speed Packet Access, the transmission rate is 14Mbits/s for downlink, while its evolution technology, HSPA+ can provide 84Mbits/s. Another change was the variety of the Internet usage. Not only the smart phones can browse websites, but also a lot of applications which use network transfer functionality have been developed. Nowadays people use their phones to browser webpages, read RSS news, and send E-mail. Additionally, many applications using the server-client architecture also make it possible for people to use their remote resources on the server and get the result on their phones. As these examples illustrate, mobile phones have undoubtedly become a complicated computer system, which has powerful hardware, the ability to do complex computation and the capacity to run an abundance of software on it.

Such changes on mobile phones have both benefits and challenges. Firstly of all, it creates convenience for programming on these platforms. For example, Maemo is an open source based platform (however, Maemo itself is not open-sourced) which is similar to Debian desktop Linux distribution. It has the Linux kernel functionality, system libraries and runtime management, which means software designers can use their programming experience on desktop Linux systems, and explore almost the whole Linux function libraries. Moreover, unlike non-smart phones, it is possible to touch some low level system functions such as Linux kernel. As a result, programming on application level, as well as on operating system level and middleware level has become possible. On the other hand, the systems have never been so powerful and complex as they are now. Handling such a complicated system is not an easy job. Managing the system resource is especially one of the most crucial tasks.

Mobile systems have their uniqueness compared to other computer systems. Firstly, robustness is highly required. Unlike desktop computer systems, a crash on mobile systems can lead to more severe consequences such as ongoing phone call drops. It is hard to imagine that mobile phone producers could persuade the customers to buy a phone that has frequent problem with making calls.

More importantly, the usage scenario of mobile systems is different than desktop computer system. For example, the network environment varies quite frequently when the people carrying the phone are moving. So it requires more delicate system resource management mechanism to handle such changes. For instance, the power consumption on a mobile phone is a crucial factor, especially with the fact that the energy technology does not evolve as fast as communications technology (a stereotype battery of today's mobile

phones is almost the same as it was ten years ago). If not handled properly, the system will waste power on unnecessary resources, such as the display, when the user is not using the phone or transmission power when the signal strength is really bad and finally, it will drain all of the energy very fast. Similarly as before, nobody would want to buy a phone whose battery can only last a couple of hours.

1.2 Objectives

Facing the challenge of system resource management on mobile phones, we aim at developing a framework to manage the resources more properly. As we mentioned, current mobile systems provide us powerful tools to design such a mechanism. In this thesis, we are aiming to design a software that can handle different situation changes and adapt the resource. We are going to introduce some of our objectives here.

We are aiming to develop a framework for context-aware resource management, where the context changes are defined using some relations of system measurements, and design the adaptations, which will schedule system resource, against such changes. For example, mobile phones have a strict constraint on power supply, which requires software developers to design advanced methods to utilize the energy, and to increase the battery life as much as possible while keeping the performance at an acceptable level. One typical case of such a requirement is to make a power adjustment based on context changes, such as traffic content or link quality. In order to perform such management, we need to monitor traffic content and link condition, and react whenever the contents and conditions change. Such adjustments include increasing or decreasing the power level, and turning on/off some hardware components. This case illustrates one typical scenario of resource management.

Our frameworks adopts publish/subscribe mechanism into event processing framework. Event processing has been widely used in such fields as financial market analysis, control systems and sensor networks. But few attentions were paid on system resource management. We provide a framework, where software designers can define the changes on different system resources and adaptations according to their requirements. Let us consider the power management example again. Our software works in a way that the software developer can define the context change (in other words, the event) he is interested in, and register the change to our software so that whenever such

a change happens, the software will make a notification. On the other hand, the software developer also needs to design the reaction when the change happens, and our software will automatically trigger such reaction when the notification arrives.

Our goal is to design software on the middleware level, so that the programmer at the application level software does not need to pay more attention to the resource scheduling, and in some of the cases, they do not even need to care about it.

We test our software using two scenarios related to energy efficiency. Naturally we will also be using some concepts of power management algorithms and terms.

1.3 Scope

Since our main target is the feasibility of event-driven resource management system, we pay more attention to the architecture level details, and some case-specific details are outside the scope of our work.

Firstly, we choose the power management for Wi-Fi data transmission as examples in this thesis, although our framework can also be used for managing the power consumption of 3G transmission. Our power adaptations are based on Wi-Fi network interface settings, but software developer can also design strategy for 3G adaptations.

Secondly, in the test case about SNR-based adaptive transmission, we only choose simple prediction algorithms for testing, since our focus is the framework that can ease the implementation of different power management policies but not the algorithms themselves. We leave the improvement of prediction algorithm out of scope.

Thirdly, we use adaptations based on Internet traffic content. We focus on TCP-based data transmission and leaves the UDP-based transmission out of scope.

1.4 Structure

The structure of this thesis is arranged as follows: in chapter 2 we review some previous work are relevant to our work. Specifically, we will introduce some articles for the concepts of mobile system resource management and

typical methods used by them, some ones about event processing for resource management, and we will compare them to each other and to our work too.

In chapter 3 we will introduce see some basic components in our work. Firstly two key factors in our work are discussed: *Publish/Subscribe architecture* and *event processing*. We will have some words about the basic ideas of them in 3.1 and 3.2. We will also talk about Maemo platform a little bit. Finally, since we mainly deal with energy efficiency problem on WLAN interface, and Power Saving Mode is going to be used in our adaptations, we will discuss about it.

The chapter 4 is about the software architecture of our work. We are going to use some UML diagrams, which provides a graphical way to model the software, and to build the architecture. Specifically, we will see the use case diagram, sequence diagram, state diagram and class diagram that make it easier to describe our software.

In chapter 5 we will see more details of our software. Firstly we will discuss about some general terms in 5.1. We have two usage cases in our work, namely, the *adaptation based on SNR measurement* and the *adaptation based on traffic content*. The 5.5 is about how we define the events and adaptations in XML. There are also some words about SNR prediction algorithms and multi-threading in this chapter.

In chapter 6 we describe the testing results of our work. As we mentioned before, there are two cases: *adaptation based on SNR measurement* and *adaptation based on traffic content*. We will discuss both cases by talking about the usage scenario, basic settings, the test results and some analysis of them.

In chapter 7 we will have some further discusses about our results. Specifically, we will propose some improvement that we can make in the future work.

Finally, in chapter 8 we will draw a conclusion about our work.

That is all about introduction to this thesis. Now we are going to step into the first part, the related work, to see some work done by other researchers and how they relate to our work.

Chapter 2

Related Work

In this chapter we will introduce some research works related to our project. We will firstly talk about some general works on mobile resource management and event processing. After that we will discuss about some papers on power management because we will use some ideas and algorithms described on them.

2.1 Resource management

As we have mentioned, our software works in middleware layer, i.e., it works between operating system and user applications. [27] discussed the basic elements on mobile middlware. For example, it talked about the basic tasks of a mobile middleware, requirements of mobile computing, some common architectures and patterns for mobile middleware, some technologies and platforms on mobile systems as well as some examples.

Context has been widely adopted into system computer systems and services. It can be used to describe the state of a system, an application or a network condition. [7] discussed about the uses of context-aware system. It introduced architecture principles of context-aware systems, and presented a layered design framework. Furthermore, it talked about some existing context-aware systems mainly focusing on middleware level. It gives us the hints on how to step into the context-aware world.

[8] proposed a context-aware middleware, called CARMEN, for mobile resource management. The main idea of CARMEN was to identify the context and the trnasitions between contexts which then trigger atomic reconfiguration of the Internet services. In order to determine the context, CARMEN

was using metadata. The metadata included such information as resource management policies, profiles for user preferences, as well as resource status for different system components.

Publish/Subscribe architecture has drawn many attentions from different areas. It is highly suitable for communications within systems which have components from different sources, and might involve remote message processing. [12], [2], [18], [10] talked about the Publish/Subscribe in many various perspectives. [12] discussed about one of the main advantages of Pub/Sub: the decoupling of the communicating entities in three different dimensions: time, space, and synchronization. [2] discussed how to find a match among all subscriptions when one event happens in a mathematical manner, which proved that the complexity in time dimension when search for one event will not be greater than $O(x^{1-\lambda})$, where λ was the number of subscriptions. It also proposed some optimization methods, which would be helpful when the number of subscriptions is huge. This paper proved the feasibility of using Pub/Sub in the manner of complexity in event matching. [18] talked about the possibility of using Pub/Sub in a distributed environment, especially for mobile devices. Because of the mobility of devices, the messages transmitted might get delayed or lost, it analyzed these situations and how to cope with them. [10] stated that designing mobile middleware using Pub/Sub can satisfy many requirements on mobile computation. In order to design such a middleware, it was fundamental to handle the scalability and change of topology from application or network. All of these articles have proved that using Pub/Sub can exactly solve the context-change handling problem in our work.

2.2 Power management

Since we are using power management as our test scenarios, we need to use some concepts and articles related to it too. [5] designed a middleware for mobile devices, which cleared some basic problems in our work. [22] proposed to use event processing in resource management. However, it only supported simple event processing. [26] proposed to use event-driven architecture on power management system. It aimed to reduce the power consumption when the mobile device is in idle mode by shutting down the device and wireless card. [28] also discussed about using event-driven framework in power management. But it only adopted it into control the core frequency of processors. Both [26] and [28] only used event-driven framework in some specific cases. But what we are proposing is to use it in generic resource management

scenario.

802.11 Power Saving Mode (PSM) [1] has been adopted as industrial standard to reduce power consumption of WNI. Specifically, it puts WNI into SLEEP mode, which will consume must less energy than normal IDLE mode when there is no network traffic going through the WNI. Many articles analyzed the PSM. For example, [31] showed that about 50% energy consumption can be saved by using PSM with appropriately setup timeout values in multi-hop wireless networks. [20] analyzed the performance of PSM setting by running a network simulation with constant web-browsing traffic. It also pointed out that using PSM introduced some performance degradation such as packet delay. As a result, it becomes a valid problem to balance the usage of PSM and normal CAM. STPM [4] discussed how to switch between PSM and CAM based on the energy saving and the performance degradation.

Many works have been done in adopting context-aware system into power management. [16] was a doctor dissertation which talked about the possibility of using context-aware pervasive computation in power management system. [29] made predictions on battery usage pattern based on existing context, such as use location traces and previous battery charge and usage patterns. In [17] it was mentioned that although context-aware power management (CAPM) has many advantages over traditional methods, it is hard to design an efficient policy, and the policy design is highly dependent on user behavior.

[30] and [13] have derived some power consumption model that we are going to use in one of our scenarios to calculate the power. [25] used SNR value as a metric to measure link quality, and it is also what we use in one test scenario. [19] used Auto Regressive Integrated Moving Average (ARIMA) to predict the SNR value.

Now that we have introduced the related literatures in our work, we will talk about some background knowledge in next chapter.

Chapter 3

Background

In this chapter we are going to talk about some background knowledge in our work. Firstly we will introduce the software platform that we are programing on. Then the Publish/Subscribe architecture, as well as event processing is discussed. We will talk about some key factors there. We used XML for configuration, event and action description. Finally, the power saving mode is used for saving energy in current mobile platform. Some of our adaptations will modify settings of it, so we will discuss about it too.

3.1 Publish/Subscribe architecture

The Publish/Subscribe (in the later context we will refer it as Pub/Sub) model is commonly used in software e.g., for messaging and event processing. The basic idea is just like the subscription to a newspaper: a user subscribes to a newspaper he/she wants to read every week by making a contract with post office. The post office makes a record of this contract with the information such as the subscriber's name, address, what kind of newspaper is subscribed, and at which frequency. As a result, every Monday, when the publisher has the latest newspaper ready and printed, the whole batch will be sent to the post office. The staff there will check all the records to find out who have subscribed to this newspaper (luckily they do not need to check it manually any more). A delivery worker driving a van from the post office will deliver the newspapers to the houses of all subscribes so that they do not need to go to the post office to check the update of the newspaper every week.

Here in this example we have mentioned three key parts in a Pub/Sub model:

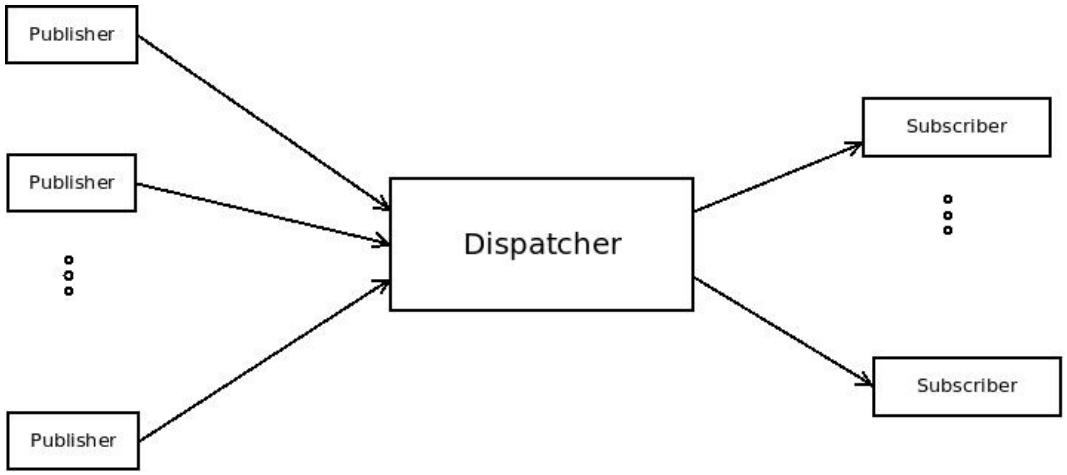


Figure 3.1: A Publish/Subscribe model

the publisher (the press of the newspaper), the subscriber (all the subscribers to the newspaper), and the dispatcher (the post office). Please see the figure 3.1. The dispatcher is the core in Pub/Sub model. It is the bridge between the publishers and subscribers. All communications must go through the dispatcher, and all publishers and subscribers deal with the dispatcher directly. In our context, the information we need to convey is the event, so we will say the event Pub/Sub in the future. Just like the post office, the dispatcher needs to know at first what events are available for the subscribers to register. The post office cannot make the contract for a newspaper that it is not aware of. So whenever a subscriber tells the dispatcher that it needs to know the occurrence of one special type of event, the dispatcher records the basic information of this subscription, including the identity of the subscriber – a process ID, thread ID, or sometimes a Globally Unique IDentifier (GUID) of the object, the identity of the event (this ID could be only in the context unique within Pub/Sub model), what kind of special requirement of the event the subscribers has, for example, some subscriber might only be interested in some special cases of an event, not all the occurrences. Sometimes some security check mechanism is performed by the dispatcher for the subscriber to make a request, and only when the publishers confirms with the request, the event can be subscribed to. During the whole subscription process, the subscriber does not deal with the publishers at all. The dispatcher here is just like an agent of the publisher, which is in charge of recording the subscription on the publisher's behalf (but we will see that the role of a dispatcher is beyond an agent, which is another SW model).

In our case, the publisher are called sensor (we will introduce them in the chapter 5). The main job of a sensor is to monitor the resource status. For example, to get the SNR (Signal-to-Noise Ratio) value frequently. The special requirement from the subscriber is sometimes delivered to the publisher too so that the publisher will analyze the monitored resource status with these requirements. Whenever the resource changes the state that fits the event description, for example, whenever the SNR value is higher than 20 dB, the sensor will send this event to the dispatcher. It should be noted that publishers do not need to know the information of the subscribers. It might need to know that some kind of event has been subscribed to the dispatcher, and for the events that have not been subscribed, it does not need to inform the dispatcher when these event happen. Also, it might need to know that there are some special requirements of one type of event. But that would be all. Publishers do not need to know who exactly subscribes to this event, and how many subscribers there are. All it needs to do is to send the event back to dispatcher. At the dispatcher, this event is analyzed, and the dispatcher finds from its records to see who have subscribed to this event. We use the plural because sometimes more than one subscribers are interested in this event, and this is actually the power of Pub/Sub model: it is a one-to-many dispatching mechanism with special requirements. The dispatcher will deliver this event to all subscribers, and they shall take their own actions against this event.

There is another benefit of such architecture: by having a dispatcher which is responsible of scheduling and managing all event subscriptions and distributions, it is possible to decouple the event producer and the consumer, and hence it is easy to add new publishers or subscribers.

3.2 Event-driven framework

Event-driven framework is another crucial factor in our work. There has been many previous work on this topic. [11] was a book about event processing. The authors introduced the basic concepts in event processing, the fields that event processing can be applied, how to fit event processing in different scenarios as well as some considerations that the software engineers need to know when they are designing their event processing systems. This book's target readers are software architects and developers, so it is helpful in practice. [24] is another book on event processing, mainly focusing on using event processing on distributed systems. It introduced how to identify event in a distributed environment, and how to route the notifications

among distributed nodes when the events happen. Complex event processing (CEP) is a layer built on top of generic event-driven architecture. It defines the complex event, which is a more complicated combination of events and more accurate abstraction of real world. CEP has some specific features and requirements, in [21] it systematically talked about complex event processing and its usage scenario in different systems. In the following parts, we will talk about some key concepts in event-driven framework.

3.2.1 Origination

In the traditional information system, for example, control system, has a procedure-based feature. It means that the basic unit in such systems is the procedure, and the system is going through all of these procedures by some pre-set order. Every system state change is predictable and the consequence is programmed beforehand. When the data amount of such changes becomes huge, databases or data warehouses are used to store such static data.

However, those procedure-based systems cannot satisfy the requirements in such fields as resource scheduling, financial market analysis or sensor networks. In all of these usage scenarios (most of them are real-time systems), there are some common and key factors:

1. The number of possible changes is so enormous that it is really hard to design procedures for every change beforehand
2. The combinations of these changes are also complex. For example, a time-ordered changes (refers to complex event) might require some specific action
3. Like in 2. a cause-effect ordered changes also requires a specific action
4. It is hard to add new procedures in the future when new changes become possible
5. The real-time data makes it hard to store and query in databases

3.2.2 General Terms

All of these factors inspired the innovation of a new system model: the event-driven architecture[11]. Unlike procedure-based model, the unit in an event-driven architecture is the so-called concept "event". An event is a special

data that represents a state change in the system. It is similar to the change in procedure-based model, but the processing mechanism is different. In an event-driven architecture, a core part is the event processing engine, which takes all the event input into its computation, identifies the target event by order or cause relations, and decides which actions should be taken against this event.

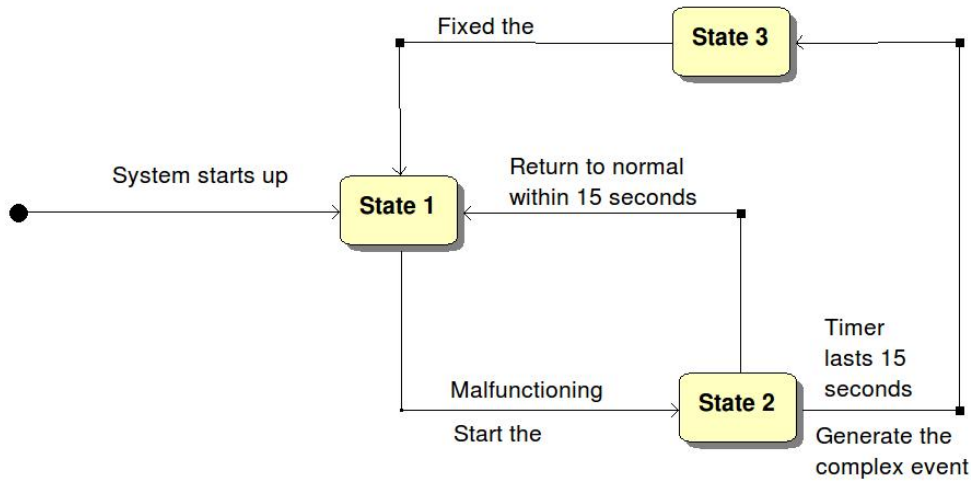


Figure 3.2: A state machine for complex event

In a complex event processing model[21], the events can be categorized into two types: the atomic events and the complex events. An atomic event is the smallest unit that cannot be divided further, and it can be detected by the simplest monitor. For example, a switch on/off of a sub-system is an atomic event. Whenever the sub-system is started up, it will send a signal to the central monitor, and this monitor decides that this event has just happened. A complex event consists of two key elements: a set of sub-event – either atomic or complex events – and a relation among these events. For example, an event that indicates the sub-system has gone into a malfunctioning state requires such sub-events: 1. the sub-system has been started up. 2. it enters into an error state 3. A timer showing how long the malfunctioning state has been lasing is launched. 3. the timer says that it has been 15 seconds and the sub-system did not go back to normal state. This complex event involves a time-order relation (the sub-system was started up at first),

and a conditional relation(only the timer lasts for more than 15 seconds this event is considered as happening). A complex event is often represented as a state machine, which describes all the involved sub-event and relations. A state changes can be either a new sub-event input or some other external triggers. Figure 3.2 shows a state machine representing our sub-system-malfunctioning event. We can see that at first the state machine is in the initial state (the black dot). Whenever the sub-system is started up, it will go to state 1, which is the normal working state. The arrow line represents a state transition, text above the line represents the condition to go to this state, and the text below it is the consequence of this transition. Whenever it has gone to the error state, it goes to state 2, and also triggers the timer. If the sub-system goes back into working state, the state machine goes back to state 1, and clears up the timer. But if the timer has been lasting for 15 seconds, it will go to state 3, and sends the signal that this complex event just happens. Using the tool of events and a complete relation description, it is possible to handle different combinations of changes, so that it will solve the problem with procedure-based model.

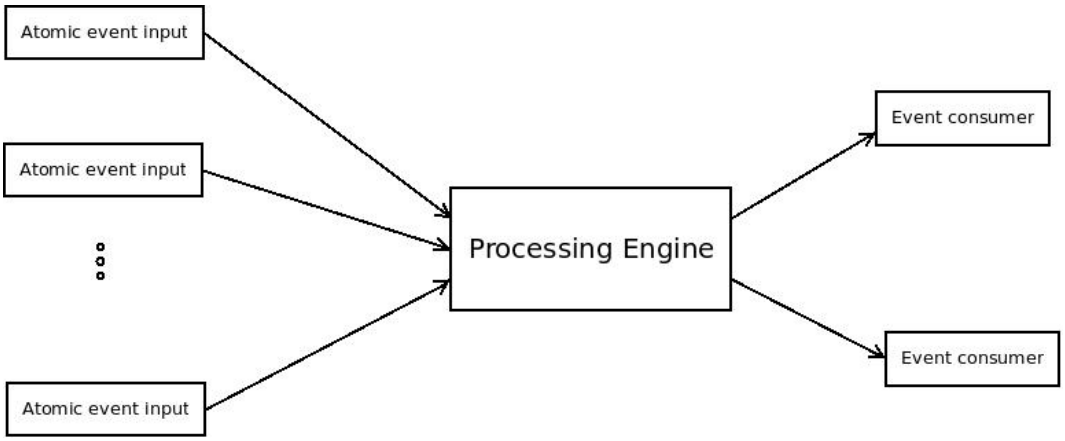


Figure 3.3: Event processing model

Figure 3.3 shows a simplest model of event processing. It involves three main parts: the atomic event input stream (can be parallel), which comes from different parts of the system, the event processing engine and event consumers. It looks similar to the model of Publish/Subscribe model that we mentioned in the section 3.1, but there are some distinguished differences too, we will illustrate them later in this section. An atomic event input gathers all the possible atomic events that the model is supposed to deal with, and they are analyzed by the event processing engine. Since the atomic

events are from different modules of the system, it is possible that multiple atomic events are input into the processing engine at the same time. So the synchronization for the processing engine is crucial. In some systems, there can be thousands of event inputs, which also requires computation speed to be fast enough. Plus, different atomic events can have different priorities, and the processing engine firstly analyzes the ones with higher priority.

After an atomic event is fed into the processing engine, it firstly checks how this atomic event will affect the on-going event computation. For example, let us see the state 2 at Figure3.2 . An atomic event will be put into the processing engine when the timer lasts for 15 seconds, and the processing engine identifies that this state machine should take this atomic event into account (there might be some other state machines that need this event too). The state machine changes its state to state 3, and a complex event is generated. There can be some more complicated relations that are hard to describe in state diagram, but in our work, we find it good enough to fit the testing scenario. We will introduce the relations in more details in the section 5.5. Here we can see the difference from the event processing model and our previous Pub/Sub model. Firstly the targets of these two models are different. The event processing model is designed to deal with the different situations or state changes, while the Pub/Sub is used to capture, and redistribute the information (message or actual event in our case). Secondly, from the implementation's point of view, a dispatcher in Pub/Sub model is part of the event processing engine, but not all of it. The event processing engine also includes the computation of complex events, which are part of the job of the publishers in Pub/Sub model.

Besides the computation of complex event, another job of the processing engine is to make action decision based on the happening of events, whether it is atomic or complex. A simple implementation of this is a rule-based or (reaction-based) model. In such model, we define the rule that when an event happens the system should obey (or the reaction the system should take). Whenever an event happens, the processing engine checks whether there is a corresponding rule in its rule library to deal with this event. The rules can be described in different formats, for example, in large scale system, a table from a database can be used to store all the rules, and whenever an event happens, the processing engine makes a query to find the rule it should use. Or in simpler systems, the rule set can be described as an indexed data structure, where the index is the event identifier. Furthermore, it is also important to describe the rules, either by the user or by system when it needs to generate the rules automatically. In order to do that, we use XML since it is easy to read for the user and to process for the system.

3.2.3 Event-Condition-Action

Event-Condition-Action(ECA) [6] structure is commonly used in event driven architecture and database management systems. It consists of three main parts as its name implies: event, condition and action. An event is the same as we have introduced, represents a context change in the system. In our software, it is the change that triggers our resource management strategies. A condition is typically a logical test. Whenever the event happens, some specific test will be evaluated, and only if this test is considered as true, the pre-defined action will be taken. An action here is the adaptation we take based on context changes. In another word, it is our resource management methods. ECA has simple pattern to implement, and it satisfies our requirements to make adaptations based on context changes.

3.3 XML

XML stands for Extensible Markup Language. It is used to process structured documents. In current information systems, the amount of data is quite huge. How to make a clear structure of these data, and how to query the correct data among the huge data flow is crucial in such systems. Besides, the Internet has become an important medium, with millions people trying to get the different pieces of data all the time, using different languages, different application software and different operation systems. So there was a need to represent the data in an efficient and standard way, so that it is easy to transfer the data among different systems. XML is an application profile of Standard Generalized Markup Language (SGML, ISO 8879)

XML files look very similar to html (HyperText Markup Language). Firstly, they are all markup languages, which have tag as the name of the element and value for the tags. Secondly, they are all structured which means that an element can have child elements. But there are fundamental differences too. The most important one is they have different purposes. HTML is designed to display contents, so that every element has their meanings. For example, headings in HTML have the tag <h1> to <h6>, and the paragraphs have the tag <p>. A user cannot create the tag by his own because that cannot be displayed correctly. On the contrary, XML is used to organize the data. As a result, the name of the tags can be anything – actually not everything can be used in one specific XML file, there is a mechanism called XML Schema to check the validity of the XML file: the names of the tags, the relations among the elements and their child elements. But it is still totally up to the

user to define which names are valid.

In our work, we use XML to describe events and rules against those events. We will talk about it in the section 5.5. Since we only use a simple functionality to organize the rules and the events, and it is going to be used only with the software's context, we did not create a XML Schema to check the validity of XML files.

Whenever we have a XML file, it is important that we can parse it correctly. There are plenty of XML parsers in different libraries and frameworks. For example, *libxml* [1] is the library provided by GNOME that is used to parse an XML file in Linux systems. In our work, we used *libxml++*, which is a C++ wrapper of *libxml*.

3.4 Maemo platform

Maemo is a Linux-based platform for smartphones and Internet tablets. It was originally developed by Maemo community[2], with most of its components being open source (although it is not totally open sourced). Nokia started to use Maemo as the platform for its products from 2007, when the 770 Internet tablet was released. The original version of Maemo is OS2005. The newest version of Maemo is Maemo 5, also known as Frementle, and it is what we are using in our work. As many cases of development on any other Linux platform, we choose C/C++ as our programming language, and we need to install the necessary software such as the SDK and compiler.

In the year of 2009, Nokia announced the first mobile phone Nokia N900 that uses Maemo 5 as the operating system. Compared to previous versions, Maemo 5 has more finger-friendly user interface, better-designed kernel functions and more friendly telephony support. Maemo platform consists of the operating system and Maemo SDK. Here is the figure 3.4[3] of the key components of Maemo platform. We will introduce some of them that are helpful in our work.

On the bottom level, it is using an ARM/OMAP-based Linux kernel based on Linux 2.6. A kernel is the fundamental part of any Linux-based functions. It is in charge of the system-level task management such process management, file system, memory allocation and dispose and the hardware cooperation (through drivers and Hardware Abstraction Layer functions). Plus, it has

[1] http://linux.die.net/man/3/libxml

[2] www.maemo.org

[3] source:http://maemo.org/intro/platform/

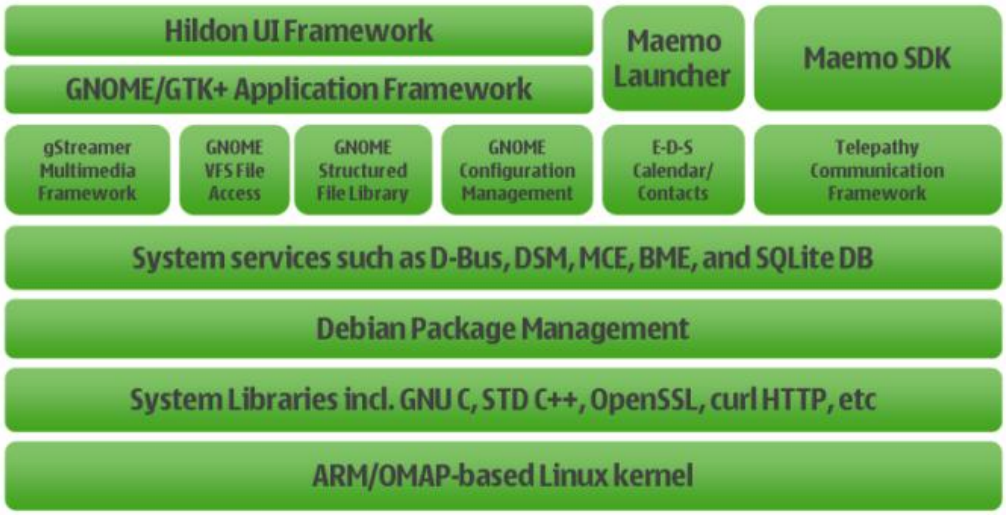


Figure 3.4: Components of Maemo platform

become the trend to include networking functions such as TCP/IP protocols implementation in the kernel, although it was not considered as part of kernel when Linux was firstly developed.

System libraries provide the necessary functions to work with the kernel at the bottom and with middleware and other framework on the top. Maemo uses standard GNU C library, meaning that most of the system libraries for any Linux system are available, and the user can deploy them just like what they did before. For example, the functions for process management, including creating a process, monitoring and terminating it are the same as any desktop or mobile Linux systems. Besides, Maemo is using OpenSSL for networking, and the standard hardware abstraction layer (HAL) library for the middleware and user-space program to communicate with the hardware.

The Maemo SDK is the part that most of the time we are dealing with. The SDK is developed using C language, and there are also C++ and Python bindings for most of the core APIs. In order to develop on Maemo, or in our case, on N900, we need to install the Maemo SDK at first. We are developing on a Linux computer with Ubuntu 9.10 distribution, and hence the installation is quite convenient and seamlessly supported. The cross-compilation environment is supported by Scratchbox. Scratchbox is an application development toolkit for compiling and configuration for different Linux systems. It is practically a sandbox environment, which isolates the target software program from the platform on which it is developed. By using a sandbox,

the target platform and development platform can be different and there will not be any harm to the development platform. Scratchbox was originally designed for Maemo and it currently supports ARM and x86 architecture systems.

3.5 Power Saving Mode

To save power consumption of the WNI, IEEE has included so-called power saving mode (PSM) in the IEEE 802.11 protocols. Regarding to PSM, there are two kinds of modes of the WNI, namely, **Power Saving Mode(PSM)** and **Constant Active Mode(CAM)**. The main idea of it is to put the device into PSM mode, which consumes much less power than CAM, when there is no network traffic transmitting on the WNI. During the PSM, much of functionality are paused, and the WNI only checks periodically if there is any traffic going through the WNI (through one kind packet called beacon packet). Compared to CAM, the power consumption in PSM is much lower, which makes the WNI always-online realistic. Figure 3.5[4] shows the comparison of power consumption when the WNI of a Nokia N800 (an Internet handset which is using a previous release of Maemo system) is in different states. We can clearly see that when the WNI is in PSM, it consumes almost as much as when the WNI is turned off. On the contrary, when the WNI is in CAM(PSMoff), it will be as high as 8 times than in PSM.

Before we illustrate how exactly PSM is working, there are two things to note: Firstly, in our discussion we are only talking about infrastructure WLAN organized by the router, the ad-hoc WLAN is beyond the scope. Secondly, when we say traffic, we mainly mean downlink traffic because that makes only sense. For example, whenever the WNI needs to send some traffic data(uplink traffic), it can go to CAM directly without any checking. But it needs this kind of checking to see if there are some packets on downlink direction.

Since we are talking about infrastructure network, within the network the PSM functionality are organized by the central router. All nodes in the network inform the router whether they support PSM, if so, they also inform about the interval they wake up. When the nodes are in sleep mode, the downlink packets (either broadcast/multicast or unicast packets) are buffered at the router, and the router announces the existence of these packets to all

[4]source:http://linuxwireless.org/en/developers/Summits/Berlin-2009?action=AttachFile&do=get&target=power-save-presentation.pdf

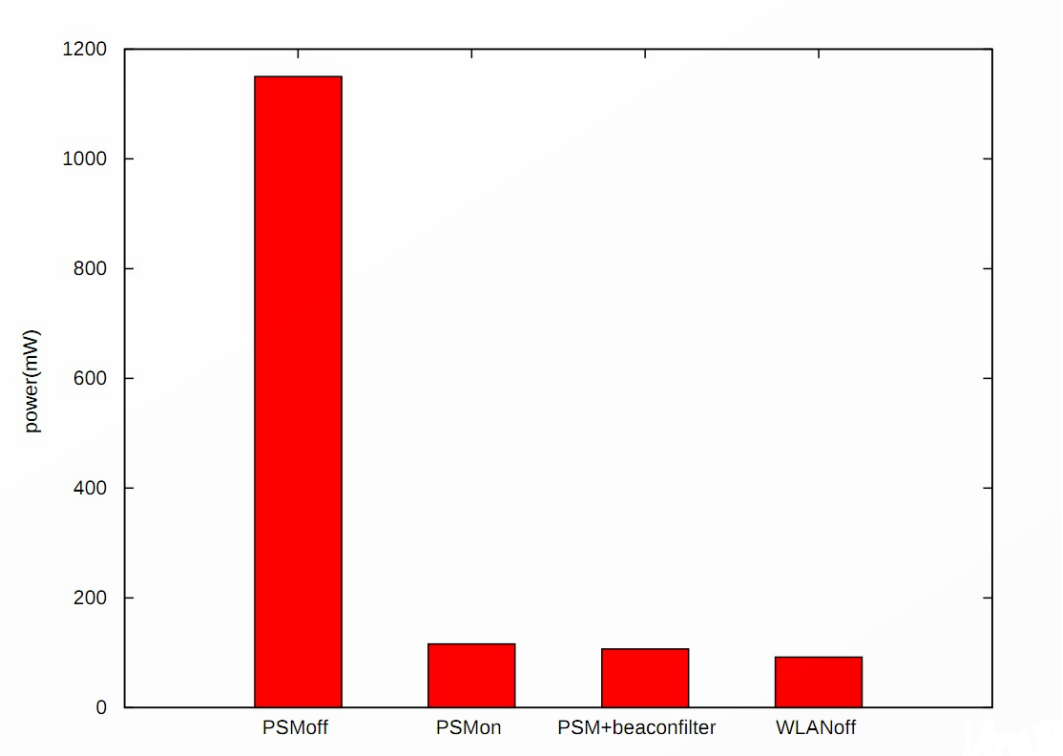


Figure 3.5: WNI power consumption in different states

the awake nodes in a special kind of packet called beacon packet. The interval between two beacon packets is set and informed to all nodes by the router. It should be mentioned that the beacon interval is not the interval for each node to wake up. The nodes wake up at their own interval, typically multiple times of beacon interval. The router sends all buffered broadcast/multicast packets right after a beacon packet, and it sends unicast packets only when the target node wakes up or when the nodes sends a request by so-called PS-poll frame. When the node is receiving packets, it stays at CAM. After successfully receiving the packets, it will go to PSM again if there is no more data.

Figure 3.6 is an example sequence diagram of a access point(AP) i.e., a router, and a node controlled by the AP. From this example, we can see that the node's wake up interval is 4 times of a beacon interval. At beacon 0, the node checks and finds that there is no packet for it, so it goes into sleep mode. After 4 beacons, the router knows that the node is waken up and hence sends some unicast packets to the node. The node stays awake to

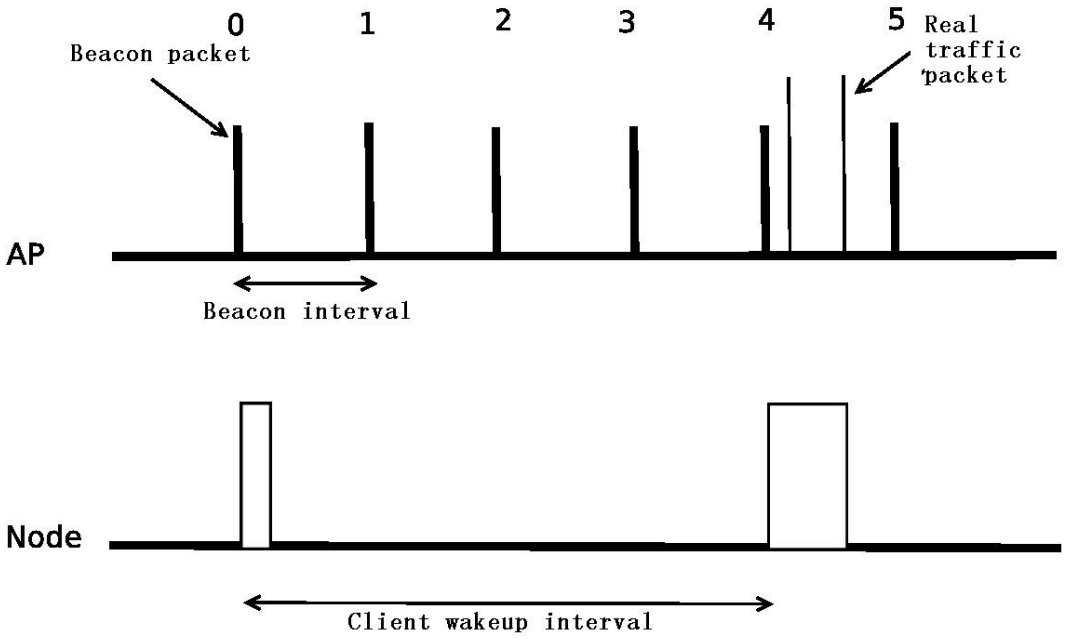


Figure 3.6: PSM sequence diagram

receive them successfully.

From our description, the reader can immediately find one important setting in this mechanism: how the WNI should behave after it receives a series of packets successfully? There are two options: it can go to sleep immediately, or it can stay awake for a certain period (called PSM timeout) waiting for possible incoming packets. The first case is called *static* PSM, while the second one is called *dynamic* PSM. It is not hard to imagine that the static PSM might save more power because it will not consume the power to keep awake for the possibly-not-coming packets. However, for obvious reason, static PSM will indeed bring latency for the transmission and hence decrease the throughput. The software developer or the user must choose carefully between these two modes and set an appropriate PSM timeout value.

Speaking of latency, there are two kinds of latencies in PSM. The first one is the time for the firmware to wake up. Usually it takes few milliseconds and can be considered ignorable. The second one is the sleeping-time and delay-to-send latency. It depends on the beacon interval and interval for the WNI to wake up. The second latency is the most significant one and will decrease the performance quite badly if the parameters are not set properly. Many papers have already discussed about this issue. We will also see the

result and make some discussions in chapter 6.

3.6 UML

UML (Unified Modeling Language)[5] has been widely used in the process of project management and development. It gives the project participators an easy way to understand what problem the software is supposed to deal with, as well as a visualized model of the design details. In modern software development area, the scale and complexity of software has been increasing dramatically. The basic unit of these software implementations is the class, which comprises of several data members and functions members, and all of these classes cooperate with each other to fulfill a specific task. Understanding how the classes are designed, as well as what the relations are among these classes is really important while not an easy job. At the same time a large scale software project requires tens of software engineers, sometimes even more than one hundred people, it is a crucial requirement for the software engineers to understand each other's design, but it is not a good idea for the software engineers to read other peers' code directly, for it is not a straightforward way to understand the architecture, especially how different classes are working together. Besides, different engineers might have their own coding styles (this is especially true for such small companies that do not have a coding style guide), which makes it even more difficult to read their code. Under such circumstance, the UML was proposed by Object Management Group in 1997, and it has been accepted by both academy as well as industry of software area and has gained huge success within short time. It is proved to be an efficient methodology in modeling of data, task and objects, and it is used through almost all the development phases: from use case analysis, requirement view, software design and implementation to software maintenance. The current UML version is 2.3 (both superstructure and infrastructure version 2.3, as well as some minor version numbers such as diagram definition version 1.0).

UML is a graphic modeling methodology for the designers and the users to understand the how the software is designed and works. Besides, it is helpful that UML can be used in different phases. As a result, there are many kinds of diagrams for different purposes. The most used ones are the *use case diagram*, *class diagram*, *sequence diagram*, *sequence diagram*, and *activity diagram*. We will briefly introduce some of them that we use widely in our

[5] www.omg.org

work. More details of UML can be found in [3].

3.6.1 Use case diagram

A Use case diagram describes a usage scenario of a software unit. This unit could be a module, a sub-system, or even the whole system. It typically includes a user of this software unit, some specific situation that this unit is supposed to deal with, and the outcome of the processing. A use case diagram only describes the characteristics of the unit seen by users or other units, i.e., it treats the target unit as a black box, no involving any implementation detail of it. Because of this feature, use case diagram is especially suitable in the software design phase, when the software designers are communicating with the stakeholders (any other people than the SW development members who will be working with the target SW), introducing how the SW is supposed to work under some specific circumstance. Of course, it also shows its power when SW developers are trying to understand what outcomes are expected when they are implementing the SW. Besides, since use case diagrams do not involve any SW implementation details, it is also suitable for general project management purposes, when the project development members are communicating with other people.

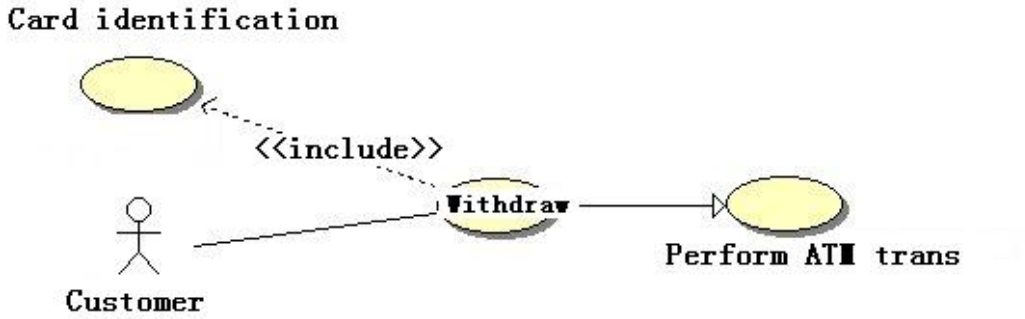


Figure 3.7: An example of use case diagram

Figure 3.7 shows a very simple use case diagram. It describes a usage scenario of a ATM module of a bank where it accepts the request of a case withdraw and takes action upon this request. The user is called customer here. It should be noted that it does not necessarily mean that the customer is a real person. This use case is very similar to a real world ATM withdraw

case, so we will not say too many words on that. The basic line is it describes the how a user will see the the software from outside of the system.

3.6.2 Class diagram

We have already known that the classes are fundamental in OOP(Object-Oriented Programming). The SW is implemented in the form of many kinds of classes and the cooperation among them. So the class diagram might be the most common diagram in UML. A class diagram describes the data members and the function members of a class, and also the relation between this class and other classes. Let us also introduce it by showing a simple example.

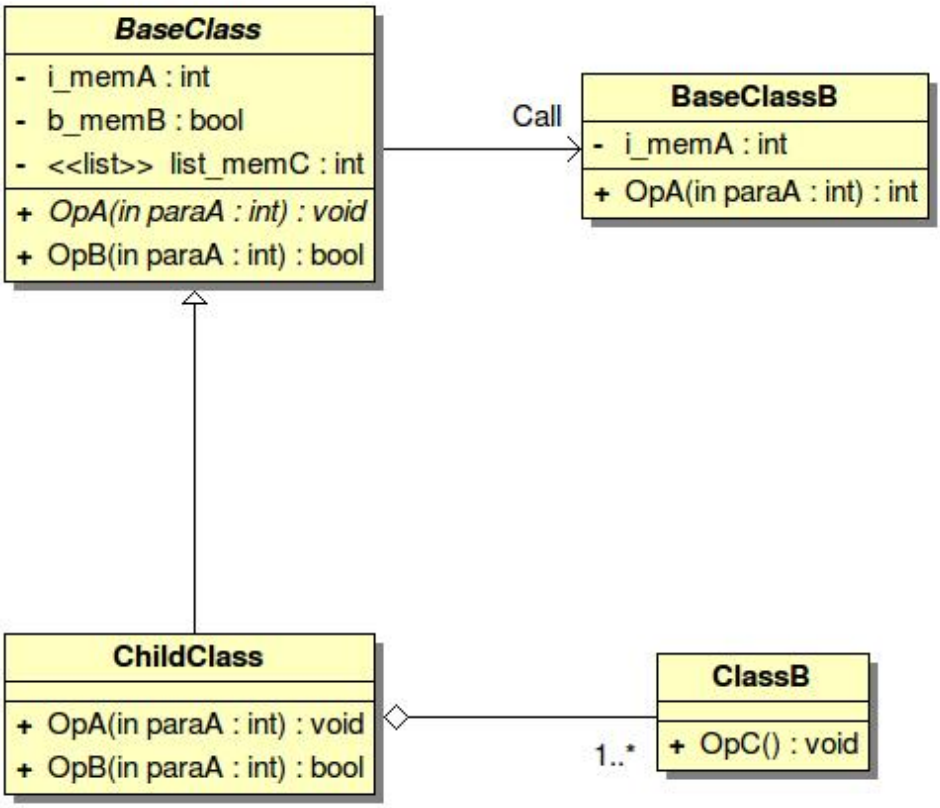


Figure 3.8: An example of class diagram

Figure 3.8 is a simple class diagram for a C++ program, but it has the most commonly used elements of a class diagram. Each block in the diagram

represents a class, with the name on the top. Above the name are data members and function members,with the plus and minus symbols indicating whether the member is public or private.

This figure also shows three relations among classes, namely, *inheritance*(a line with a triangle at one end), *directional association* (an arrow line ->), and *Aggregation* (a line with a hollow diamond at one end). Inheritance is very crucial in object-oriented programming, and the user must already be familiar with the concept. An association is a general relation linking two classes. It means that there exists some connection among these classes, but a general association does not say what kind of connection it is. Finally, an aggregation represents a "has a(or many)" relation. The end with a diamond points to class that is the container. In our example, the diagram shows that an instance of class *ChildClass* can have multiple (1 to infinite) instances of class *ClassB* as its member (the number *1...**, which is called *multiplicity*, means there can be at least one target class instance, and at most infinite instances). There are some other elements in class diagram, for example, an association called *Composition* which is similar to *Association*. We will not going to introduce them, and users can refer to [3] for more details.

3.6.3 Sequence diagram

A sequence diagram draws a picture of how the different parts of the system are cooperating with each other, and how the process is on-going and in what order. Normally it shows the order in time sequence. A sequence diagram typically shows the activity of one use case. Hence there should be several sequence diagrams in the whole project. However, in our work, we found one diagram with some fundamental classes involved is already enough to show how our software is working. Besides, since the sequence diagram of our SW is actually quite straightforward, we will show it to introduce the elements of a sequence diagram. But we will discusses about the actual sequence flow in the chapter 4.

Figure 3.9 is the actual sequence diagram we used. On the top of the diagram there are class names that involved in the use case. They are not necessarily all of the classes used in the processing. Instead, we only show some important ones. As we said, sequence diagrams show the time order.

The time is described in the diagram from top to bottom. In other words, time is going through from top to bottom. Below every class there is a bar, which is called activity bar. The length of an activity bar shows the lifespan of one class. For example, we can see the start of the activity bar of class

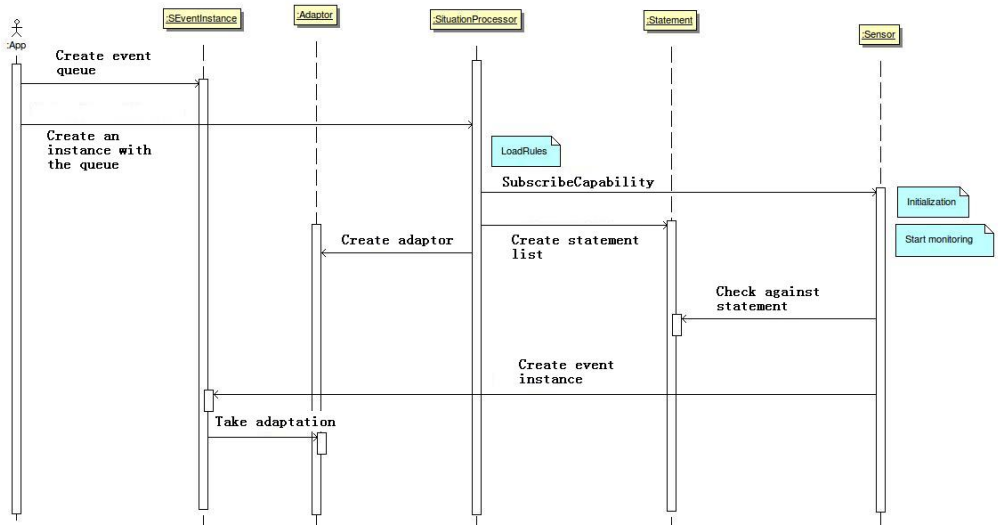


Figure 3.9: A sequence diagram

Sensor is lower than the one of class *SituationProcessor*, it means the *Sensor* is created after the *SituationProcessor* is created. Whenever a class object is deleted, the activity bar ends. But in this diagram, all of the classes are deleted in the end of the running. No one gets deleted in the middle of the running.

The arrow line between two class activity bars represents the message, i.e., contents of the cooperation among classes. For example, the first message of all is the *Create event queue*. It means the application main program creates a queue (implemented as a list) of SEventInstance. A line with an arrow is called an asynchronous message. It means the message sender does not need to wait for confirm of this message from the receiver (it is likely that there is no confirm at all). Another type of message is synchronous message, meaning that the sender must wait for confirm from the receiver. In the diagram, a synchronous message is represented as a line with a filled square.

3.6.4 State diagram

A state diagram is used to describe the behavior of a system. Such behavior is represented in the form of state transitions. To what state the system transits depends on some system inputs and current state. A state diagram can be used to represent a finite state machine. More description and one example of state diagram can be found in the section 3.2.

Above is some background knowledge. In the next chapter we will introduce our software architecture.

Chapter 4

Software Architecture

In this chapter we will firstly introduce the user with some basic tools and methods that we used in the project development. Namely, UML for software management and design. After these concepts, we will officially start to talk about our software implementation by presenting the software architecture.

In this chapter we describe the software architecture in our work. Specifically, we will illustrate it by showing some UML diagrams that we described in 3.6. It should be noted that only structure level information will be presented here, the implementation details will not be introduced. Instead, we will talk about them in the chapter 5.

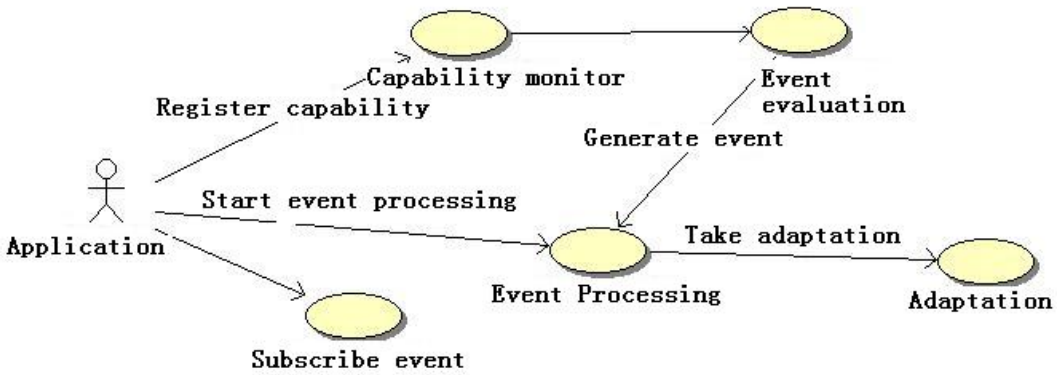


Figure 4.1: Use case diagram

Figure 4.1 shows the use case diagram of our software. It includes five sub use cases(not including the releasing phase), namely, *Subscribe event,Capabilities*

monitoring,*Event processing*,*Event evaluation*,*Adaptation*. At start-up, the applications firstly loads the rule definitions, finding out what event and capabilities (we will introduce the capabilities also in the chapter 5) the user is interested in. This is not showed in the diagram since it is too simple to be called another sub use case. After that, the SW subscribes to the event and registers the capabilities, then the event processing is started up. At run time, the sensors check the newly updated capabilities and evaluate them against the event definition. Whenever a pre-defined event happens, it inserts the event instance to the situation processor. The situation processor will check the event and adaptation definition, making the adapters to take the adaptations. That is the main idea of our software. We are not going to show the sub use cases since it will involve too much details here. But we can see more about it in the sequence diagram.

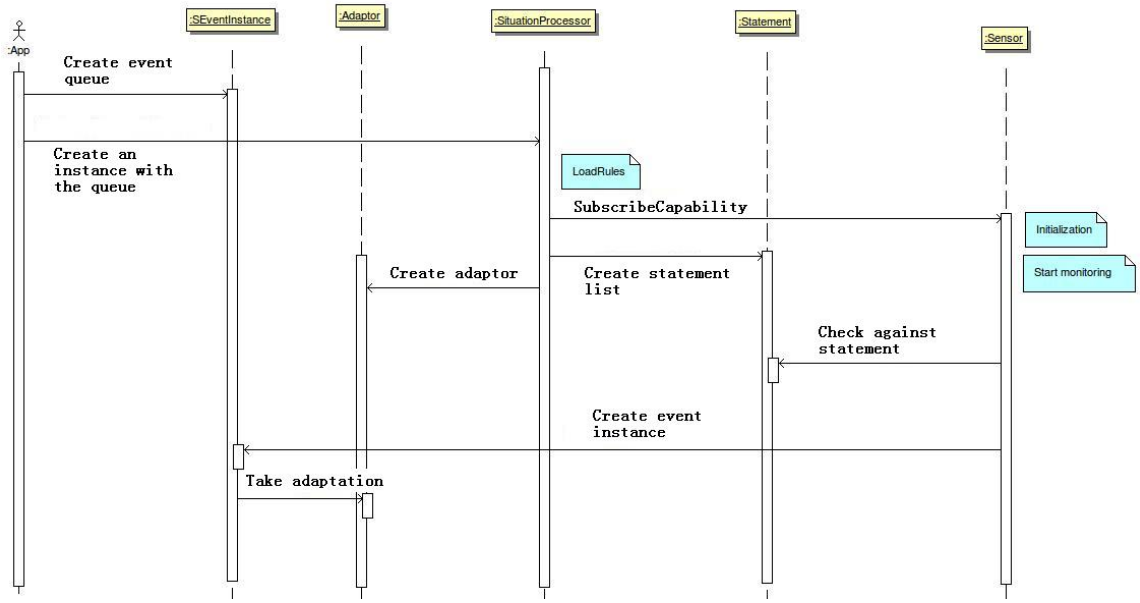


Figure 4.2: Sequence diagram

Figure 4.2 is the sequence diagram of our work, from which we can see more details. The application firstly creates an event queue of the type SEventInstance, using it to initialize an instance of SituationProcessor. The SituationProcessor loads rules written in a XML file. The Statement and Adaptor are created based on these rules and relevant capabilities are registered. After that, the events are subscribed. All the sensors check the capabilities and use them to calculate the event. Whenever an event happens, they will generate a new SEventInstance object and insert it to the event queue. The

SituationProcessor is processing the events in the queue in another thread. It will check event-rule definition and make the adaptors to take adaptations.

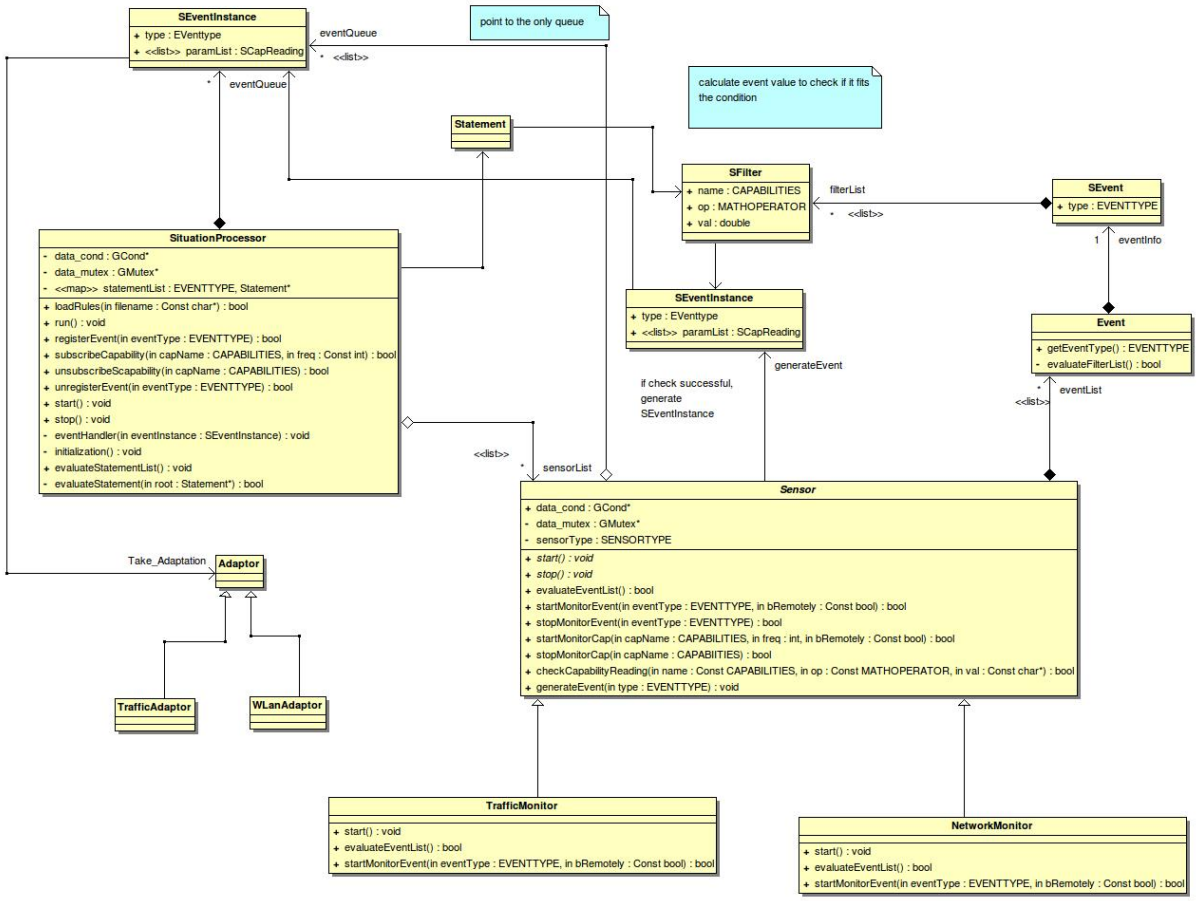


Figure 4.3: Class diagram of the software

Figure 4.3 shows the architecture and relations of all important classes in our work.

The class *SituationProcessor* is the core of our software. We will leave out the implementation details later. But here we shall introduce some functions. As we mentioned before, the SituationProcessor loads the event and rule definitions in a XML file. It subscribes to the event based on these definitions. Besides, it has an event list with the type of *SEventInstance*, which stores all the events to be processed. The SEventInstance has a pre-defined EVENTTYPE to indicate what this event means. SituationProcessor initialized the event by some *Statement* instances. All of these instances have a list of *Fil-*

ter, which defines the condition that the event is considered as happening. Currently all of the conditions are value-based, i.e., they are some arithmetic calculations. Having a list of such filters means that we can have multiple conditions at the same time and have some logical calculation among these conditions. Currently we only support logical AND calculation. All of these conditions must be satisfied so that the event happening is considered as true. A *Sensor* is the object that is actually monitoring the system resources. It is the base class and it now has two derived classes: *TrafficMonitor* and *NetworkMonitor*, which measure the traffic-related resource and network-related resource respectively. The *Sensor* has some common interfaces. For example, to start and stop monitoring the event, the capability, to evaluate the event list (each sensor can be registered with several events) and to generate the event. Whenever an event happens, the sensor will create an instance of *SEventInstance* and insert it to the event queue. All of the sensors share the same event queue, which is accessible to the *SituationProcessor*. The *SituationProcessor* is processing the events all the time in another thread than the main program, in a First-Come-First-Served manner. We can also add the priority for all the events in the future, so that it will classify these events by different priorities. As soon as an event is processed, the corresponding *Adaptor* will take the adaptation according to the event-rule definition. We now have two kinds of adaptors, *TrafficAdaptor* and *WLanAdaptor*. *TrafficAdaptor* is responsible for modifying some TCP transmission parameters to manage the resource, while the *WLanAdaptor* is responsible for changing the setting of the WLan interface.

We have seen the architecture of our software. In the next chapter, we will step into more details of the implementation hence the user can understand how the SW works more deeply.

Chapter 5

Implementation

In this chapter I will talk about more details in the implementation of our software. These details are helpful for the user to deeply understand how our software works, and they will also make it easier to read the next chapter which describes the test results and analysis.

5.1 General terms

In this section we are going to introduce some basic objects used in our software. They are used for both testing cases, so that it will be easier to discuss the specific implementations when the user has read this section.

Two main classes in our system are SituationProcessor and Sensor. A SituationProcessor is what is called the event processing engine in event processing's world. We will talk about the SituationProcessor in the section 5.5, so now we want to say some words about the Sensor. A sensor is an object that is able to monitor and generate a series of related events. For example, in our network monitor case, the events are such basic event as getting monitored SNR and predicted SNR, and some complex events such as SNR value change of HIGH_TO_LOW_SNR(each sensor has a list of such complex events and is evaluating them all the time until being told to stop). Figure 5.1 shows the class diagram of Sensor.

In order to explain how the sensors work without any ambiguity, we need to introduce two basic concepts at first.

The first one is the capability. A capability is a basic event that our software can monitor without any complex event processing and logical calculation–

Figure 5.1: Class diagram of Sensor

both might end up producing some complex events. In some literatures they are called atomic event. Examples of such capabilities are SNR values, service set identifier (SSID), power saving mode, packet interval and size, etc. All of these capabilities can be got by directly quarrying for example the operation system or the hardware abstraction layer (HAL). They are the basis of our event processing mechanism, for all of the complex events and actions (adaptations) are generated based on them. In our case, the most important capabilities we are going to use are the monitored and predicted SNR values. It is also possible to set the frequency to check the capabilities for each sensor.

The second concept is the event. In the rest of this section, the term "event"

refers to a complex event. In another word, it is produced by some logical calculation on some basic events (capabilities changes) or some other complex events. We have defined ten such events: LOW_TO_HIGH_SNR, HIGH_TO_LOW_SNR, NEW_PACKET, NEW_CONNECTION,NO_CONNECTION, ADD_CONNECTION, DELETE_CONNECTION, NEW_BURST, END_BURST, WRONG_END_BURST. But let us firstly go back to the general properties of the event. An event has a list of related capabilities. At runtime, the sensor is updating the capabilities and evaluating the events according to the set frequency.

Now let us get back to Sensor class. We have several explanations.

1. The eventQueue points to a list that is commonly shared by all the sensors and the SituationProcessor. Whenever an event is evaluated to happen, the sensors just inserts this event (the function *generateEvent()* is used)to the eventQueue, and the StuationProcessor is processing all the events in this list in a first-come, first-served(FCFS)manner.

2. The eventList stores all the registered events that this sensor is supposed to evaluate, and specifically under what situation these event can be considered to happen. To do the logical calculation, we defined the class SFilter, which stores the names of capabilities and the type of calculation as well as a double type value against which the calculation should perform(Currently only numerical calculation are possible). Six kinds of calculations are supported: $\geq$, $>$, $\leq$, $<$, $=$ and $\neq$. And we only support *AND* calculation among all the filters, i.e., only if all the calculations are true, the event is considered as happened. In the future it is expected to support more logical calculation among the filters, and other types than numerical calculation for each filter.

3. The virtual function *checkCapabilityReading()* is used to do the calculations for each filter by the set frequency.

4. The *GCond* data_cond* and *GMutex* data_mutex* is a mutex lock used for multithreading synchronization. We will discuss about it in the section 5.6.

5. There are also some functions to get or set the supported capabilities for the sensor, and to do the initialization.

As mentioned before, there is an event processing engine, whose main responsibility is to manage all the sensors, as well as to take the adaptations when the event happens.

We have drawn the basic picture of the sensor. Now let us see some inheritance classes of Sensor, namely, NetworkMonitor and TrafficMonitor.

5.2 Network monitor

In this part we will describe the network monitor. The main purpose of a network monitor is to get the link quality all the time and provide the information on which the process engine based to decide whether some specific adaptations should be taken on or not. In another word, if the network monitor cannot provide correct information, the target actions, no matter how delicately they are designed, will not get the expected affects.

Like we have stated in the introduction chapter, the mobile devices have encountered more challenges than any other type of network terminals. Among these challenges, two most important ones are the environment effects and the mobility of the device. Unlike wired network, which has a relatively stable and easier-to-analyze transmission characteristic, the wireless transmission is much more complicated. Depending on different environments, such as indoor or outdoor, metropolitan or suburban, different channel models have been proposed. The book [15] analyzes the these modeling methods. But we must admit that these models are far more away from perfect–if there would be any perfect model. All of them are approximations to the actual cases under some certain conditions. Whenever those conditions change, the performance of the model deteriorates or even becomes unacceptable. To make the problem trickier to handle, the devices themselves could be being moved all the time, and the velocity can vary quite dramatically. Typical scenarios are pedestrian and vehicular cases, which need of course different channel models.

To handle this complex situation, we must resort to some other way to manage the resource besides trying to find a precise channel model to predict every change the mobile device has been going through. The first step is to find out a metric to the link quality and all of our adaptations are taken against this measurement. Many papers have discussed about this topic. [25] points out that strong signal reduces energy cost and signal varies by location. So the value of signal-to-noise-ratio (SNR) can be considered as a good measurement to the link quality. Now we must find out a way to measure the SNR value in our software. One announcement we need to make is, although we have been talking about general SNR, but in our software we only measure the one that the WLan adapter (the WNI) receives. The 3G

signal is our of the scope of this thesis as we mentioned on section 1.4.

Fortunately there is a system call function on Linux to get the SNR value as well as some other system measurements. A system call is the function a userspace application uses to communication with the kernel. The function we use is called *ioctl()* [1], which stands for input/output control. Since we are going to use ioctl in many cases in this thesis, we want to say some more words about it.

As we know, in Unix/Linux system, the kernel can manage the resources as if they were some files (which are called virtual files). By using the similar way to manipulate the file, we can also get or set the parameters of a device or the system. The *ioctl()* is based on this very assumption to control the devices. The syntax is:

```
int ioctl(int d, int request, ...);
```

The parameter d is an opened file descriptor, and parameter request is a device-dependent request code, which we will discuss more later. There is a third parameter, an untyped pointer to a block of memory. The return value indicates whether the operation is successful or not, on success it returns 0 and on failure it returns a negative value. In order to get the SNR value (in fact, to get a signal-related structure which contains many measurements), we call the *ioctl()* like this:

```
if (ioctl(skfd, SIOCGIWSTATS, &wrq) < 0)
{
  perror("ioctl SIOCGIWSTATS");
  return(-1);
}
```

As we said already, the skfd is a file descriptor which has already been opened. SIOCGIWSTATS is a pre-defined control code, whose purpose is to get (G for get)the power-related statistics. wrq has the definition as:

```
struct iwreq wrq;
```

This is a common structure which is defined in the file "wireless.h". By common we mean it has a union member of type iwreq_data. The type iwreq_data has a struct iw_point member, which points to the different data structures for different sets of parameters. As a result, iwreq type be used to store information for different sets of parameters–but since it is a union, only one set exists at one time.

[1] http://www.kernel.org/doc/man-pages/online/pages/man2/ioctl.2.html

For our SNR monitor case, we use an iw_statistics and assign it to the iw_point member (cast to (char*) at first). The iw_statistics has the definition:

```
struct  iw_statistics
{
  __u16 status; /* Status
  * - device dependent for now */

  struct iw_quality qual; /* Quality of the link
  * (instant/mean/max) */

  struct iw_discarded discard;
  /* Packet discarded counts */

  struct iw_missed miss; /* Packet missed counts */
};
```

and struct iw_quality has the definition:

```
struct  iw_quality
{
  __u8 qual; /* link quality (%retries, SNR,
  % missed beacons or better...) */
  __u8 level; /* signal level (dBm) */
  __u8 noise; /* noise level (dBm) */
  __u8 updated; /* Flags to know if updated */
};
```

The SNR value can be derived by subtracting the noise level from the signal level.

Now let us step into more details. Firstly we should establish that there will be only one class (we call this class WLANAdaptor) instance which is monitoring the SNR value. The reasons are: 1. Only one instance is needed by even multiple program modules, so it is a waste to instantiate more objects to do the same thing. 2. There will be conflicting values that we get from multiple instances. To ensure such an unique condition, we are using the design pattern *Singleton* which described in [14]. Specifically, we specify the only constructor of the class WLANAdaptor as private function, preventing it from being constructed by any user program. Whenever we need to use this class, we call the static function *WLANAdaptor::getInstance()* to

get the only global instance of this class. The WLANAdaptor is in charge of getting and setting some WNI parameters. However, it is not the network monitor (called NetworkMonitor class) itself – or at least not all of it. The actual network monitor keeps records of what modules have subscribed to some specific network-related complex events, for example, the event *SNR_BECOMES_BAD* (this is not an event in our software, just an example). Additionally, it also schedules to get some atomic events to measure these complex events. For instance, the network monitor gets the SNR values at a pre-defined frequency and every time it gets a new value, it calculates the *SNR_BECOMES_BAD* event against this value and probably other parameters. The reader can see the difference of the functions of NetworkMonitor and WLANAdaptor: the NetworkMonitor gets some network measurements by calling the functions of WLANAdaptor, and the NetworkMonitor calculates the events, and takes actions when some special events happen. It is again the WLANAdaptor's job to do the actual actions for WNI. All of the ioctl function calls we introduced before are done by the WLANAdaptor.

(a) WLANAdaptor (b) NetworkMonitor

Figure 5.2: Class diagrams of WLANAdaptor and NetworkMonitor

Figure 5.2 shows the class diagrams of WLANAdaptor and NetworkMonitor.

All of the get_* and set_* functions in WLANAdaptor either get or set some specific network parameters (of course, some parameters can only be read, not set, like the SNR value). Most of them are using the ioctl function. The functions *get_WLAN_timeout()* and *set_WLAN_timeout()* are not used by NetworkMonitor, but the TrafficMonitor that we will introduce in the next section.

The NetworkMonitor has one fix length array to store monitored SNR and another one to store predicted SNR. The predicted values are calculated based on the monitored SNR values. We will discuss about the prediction algorithms in the section 6.1.1. We have already mentioned that two complex events are used in NetworkMonitor:LOW_TO_HIGH_SNR and HIGH_TO_LOW_SNR. A LOW_TO_HIGH_SNR event happens when the previous monitored SNR value is lower than a threshold (therefore it is considered as low SNR value) and the predicted SNR value is higher than this threshold. A HIGH_TO_LOW_SNR event is the reverse. Needless to say, we need to define the threshold. In fact, this value should reflect the trend of the SNR changes. In another word, it will be varying all the time with respect to the signal strength change. So we need to update this value after we have got an amount of SNR samples. In practice, we do this update whenever the SNR array is full(15 samples in our test). There are two measurements that we take into account when deciding the threshold: the mean value of the SNR samples and the variance of them. When the mean value is high enough, and the variance is low enough, we say that the SNR values are stably high, so the threshold should be some high value. If the mean value is low, naturally the threshold will be set at a low value. There is a third situation: the mean value may be high, but the variance is also high, it means the SNR values are fluctuating quite often, and in this case, we should also set the threshold at a low value. In our test case, the threshold is decided by the formula:

$$SNR_{threshold} = \begin{cases} 15 & (\mu \leq 20) \vee ((\mu > 20) \wedge (\sigma \geq 5)) \\ 20 & (\mu \geq 20) \wedge (\sigma \leq 5) \end{cases} \tag{5.1}$$

where the μ and σ are derived by

$$\mu = \frac{1}{N} \sum_{x=t-N}^{t-1} SNR(x) \tag{5.2}$$

$$\sigma = \sqrt{E[(SNR(x) - \mu)^2]} \tag{5.3}$$

We will verify the rationality of this formula in next chapter when we analyze the tests.

When the SNR value is predicted to be low, more specifically, when the HIGH_TO_LOW_SNR event happens, we want to pause the TCP transmission since it will consume more energy to transmit in a bad link quality. In order to do that, we can set the TCP receive window size to zero. Whenever the signal is strong again (event LOW_TO_HIGH_SNR happens) we restore the window size to its previous value before setting to zero. We will talk about how to set the window size in the section 5.3 because we will have the necessary knowledge by then. But before that, we shall at first say some more words about the TCP receive window itself.

The TCP receive window is working like this: the receiving side of a TCP connection specifies an amount of data (in bytes) it can buffer at one time before it sends an acknowledge back to the sender. In another word, the sender can only send this amount of data cumulatively, and if this amount of data has been sent but the sender does not get the acknowledge packet (when the sender is also acting like a receiver, this packet usually comes with the sending data in the other direction. This is called piggybacking) from the receiver, the sender will pause until such packet arrives. Any time the acknowledge packet is received, the sender will clear the sending counter and start it over again. It should be noted that the window size can be negotiated and changed during transmission. In fact, changing the windows size is often used in congestion control [23], [9], many advanced algorithms to update the window size have been discussed. In this thesis, we use the fact that if the window size is set to zero, the sender will not be sending any packet at all until the window size is changed to any positive value, but the TCP connection itself is still maintained until a timeout or it is released by the receiver. By setting window size to zero and restoring it to the previous value before zero, we manage to pause and resume the TCP transmission. We will step into the implementation of setting window size in the section 5.3 because only by then we will have enough details.

So far, we have introduced the necessary parts for network monitor, we will show the test result and analysis in chapter 6. But before that, we shall go through the implement elements for traffic monitor as well as event rule definition and evaluation.

5.3 Traffic monitor

In this section we will talk about traffic monitor. A network monitor keeps watching over the link quality, while a traffic monitor listens to the Ineternet

traffic content on which it makes adaptations based. As we mentioned in 1.3, we are only discussing about TCP traffic, since it is one of the most used protocols in the Internet and the pattern is relatively easy to analyze.

However, there are still many kinds of TCP usages, and they differ from each other quite dramatically. For example, web browsing traffic is one of the most important TCP usages which has its unique behavior pattern. After a TCP connection is established and the user has requested for the web page data, the server will send all data in less than 200ms with an amount of data of less than 10 megabytes. After that, this connection will be closed(to simplify our discussion, we do not consider the advance technology that predicts and maintains the connection for potential future request). It might be simple to analyze from some point of view. On the other hand, it is quite hard to predict the pattern at local host because: 1. different web pages have distinct contents and data amount 2. when a user starts a web browsing connection is random. Hence we can characterize web browsing traffic as transient and light loaded, at the same time also hard-to-predict. A second type of TCP use is the interactive traffic. For example, Secure SHell(SSH) is a protocol which is built on the top of TCP, which allows two devices to exchange data through a secure channel. In order to maintain the secure shell, there is quite much TCP traffic. The amount of such kind of traffic is relatively small. Arbitrary adaptation may cause some problem to the connection. For example, it is reported [2]that when using SSH with PSM on, the connection may be dropped because the maintenance packets are buffered at the router and hence the terminals might assume the connection is failed. If adaptations are expected to take on such kind of traffic, they should be designed protocol-specific. The third type of TCP traffic is streaming or file transfer such as FTP. This type of traffic is normally large in amount and is a constant flow. It takes huge bandwidth and the Internet usage time. So it is useful if we can design some generic adaption mechanism for them. There is a small difference between streaming and file transfer: nowadays streaming usually uses traffic shaping technology. Specifically, instead of sending the whole bulk of traffic until it ends, the server sends the traffic in the unit of burst. A burst is a series of TCP packets which are continually sent, and the interval between two burst are usually much larger than the packet interval inside a burst. The server is using different shaping algorithms (dividing the whole traffic into bursts using different sizes, intervals and rates). It will optimize the throughput and improve latency by using shaping technology.

Our main focus is on the third type of TCP traffic, especially the one that

[2]http://wiki.maemo.org/Wifi_power_saving_mode#Routers_known_to_be_incompatible_with_PSM_mode

is using traffic shaping. So we must get the information of each TCP connection: the packet size, packet interval as well as burst size and interval. After we gather all of the information, we use them to analyze and predict the TCP traffic pattern, make some adaptations when there are light traffic load.

Let use first check the class diagram of TrafficMonitor.

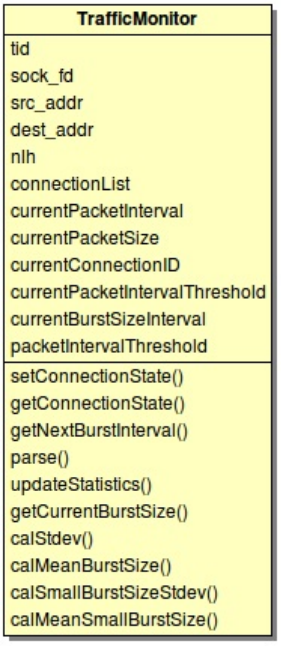


Figure 5.3: Class diagram of TrafficMonitor

Figure 5.3 is the TrafficMonitor class structure. For simplicity reason, we only show the essential members and operations of this class. A TrafficMonitor is maintaining a map from an integer value (we call connection ID) to a struct which has the information of the connection. The declaration is

```
std::map<int, SConnection> connectionList;
```

and the definition of SConnection is

```
struct SConnection
{
  int connectionID;
  int burstCount;
  int packetCount;
```

```
  int currentBurstSize;
  int lastBurstSize[10];
  double lastBurstInterval[10];
  double predictedBurstInterval[10];
  int packetInBurstCount[10];
  int packetSizeThreshold;
  double packetIntervalThreshold;
  int burstSizeThreshold;
  double maxPacketInterval;
  int minPacketSize;
  bool bPredictable;
  int predictedIndex;
  int state;//0:idle;1:active;2:unknown.
  struct timeval intervalStart;
};
```

In this struct, it stores the state of the connection, some counters of burst and packet, as well as burst size and interval. Besides, there are two thresholds values to decide whether a packet belongs to a burst and whether it is the last packet of a burst. We will introduce firstly how those thresholds work. A packet interval threshold means that whenever the interval between two successive packets is smaller than this value, they can be considered as belonging to the same burst. So when a packet arrives and our software makes the determination that it belongs to the current burst, the *currentBurstSize* will get updated by adding the size of this packet. If the packet interval is larger than this threshold, it will assume that the packet belongs to a new burst, and is will also update the statics accordingly. When the *currentBurstSize* exceeds the *burstSizeThreshold*, we also assume that this second packet is the last packet of the burst. We can see that the values set for the threshold are important, if they are not set properly, our whole analysis will be not precise. As we mentioned before, the shaping algorithms vary from each other for different TCP servers. Training is a good way to get a proper value. But then our software would be highly depending on the training and not generic. Besides, quite many servers change the shaping algorithm at run time. So we make our software update the burst size threshold value on-line. It is calculated as the average value of a series of last bursts. By doing so, we make sure that our burst size threshold follows the change of traffic shaping.

The functions *calStdev(), calMeanBurstSize(), calSmallBurstSizeStdev(), calMeanSmallBurstSize()* are used to get statics for analyzing and they are quite straightforward, so we will skip the details of them.

In order to get the individual packet information (as well to set the TCP receiving window size for the network monitor case), we are using netfilter. Netfilter is "software of the packet filtering framework inside the Linux 2.4.x and 2.6.x kernel series." [3] It resides in kernel space and provides the hook (a hook is normally a call-back function when some specific condition is satisfied) to intercept and manipulate packets.

We designed a Linux kernel module using netfilter. So firstly we shall say some words about the module structure. A kernel module program is usually written in C. The basic functions and definitions are in three files: *linux/-module.h, linux/kernel.h and linux/init.h.* So in the main file they should be included. The simplest module includes two steps: defining initialization and exit functions (in our case, we call them *ec_module_init* and *ec_module_exit*) and registering of them. Such functions are normally declared as static and registered using the defined macros *module_init(ec_module_init)* and *module_exit(ec_module_exit).*

In the *ec_module_init* function, we initialize the necessary variables, then create a socket used for communicating with our user-space program (we will introduce this later). We also register the netfilter hooks using *nf_register_hook().* As we mentioned before, a hook is a call-back function. This function requires a struct *nf_hook_ops*, which has a definition as:

```
struct nf_hook_ops {
        struct list_head list;

        /* User fills in from here down. */
        nf_hookfn *hook;
        struct module *owner;
        u_int8_t pf;
        unsigned int hooknum;
        /* Hooks are ordered in ascending priority. */
        int priority;
};
```

This structure stores such basic information as priority(*priority*), protocol family(*pf*, PF_INET for IPv4), the actual function(*hook*) as well as the hook number(*hooknum*)–in netfilter the hooknum is defined as the function types. We shall use *NF_INET_LOCAL_IN* standing for incoming packets and *NF_INET_POST_ROUTING* standing for outgoing packets. We defined the actual functions

[3] http://www.netfilter.org/

```
static unsigned int hook_local_in(unsigned int hooknum,
    struct sk_buff *skb, const struct net_device *in,
    const struct net_device *out, int(*okfn)
    (struct sk_buff *))

static unsigned int hook_local_out(unsigned int hooknum,
    struct sk_buff *skb, const struct net_device *in,
    const struct net_device *out, int(*okfn)
    (struct sk_buff *))
```

After these registrations, whenever an IP packet is coming into or going out of the device, the corresponding function will be called and we can get the information and make adaptation (specifically, setting the TCP receiving window size) as we want. Such information includes the IP address and port number (used for identifying connections), protocol type (we are only interested in TCP), packet size and arriving time (used to calculate the interval). But we still cannot use the information directly from netfilter, since it is running on kernel-space. We need to pass the packet information to our user-space program. In order to do that, we are using netlink[4]. Finally, in the exit function, we release the resource (socket number, for example) and unregister the hooks.

Netlink is a IPC (Inter Process Communication) solution to communication between the kernel space and user space (of course, it also makes it possible to communicate between user space processes). It is socket-like, meaning that the user firstly creates a socket which is user-defined specifically for Netlink to use. As soon as the socket is opened, the two processes can send messages on it. Netlink is using the AF_NETLINK socket family.

There are some pre-defined Netlink protocol subset. For example, NETLINK _ROUTE family makes it possible to communicate with the kernel module which is in charge of the IPv4 routing. Hence, the user program is able to get the IPv4 route table update as well as to modify some routing table parameters and queuing disciplines. Here is a piece of code snapshot of those pre-defined protocols in the netlink.h header file, typically located in the */include/linux/* folder.

```
#define NETLINK_ROUTE      0 /* Routing/device hook     */
#define NETLINK_UNUSED     1 /* Unused number           */
#define NETLINK_USERSOCK   2 /* Reserved for user       */
                             /* mode socket protocols   */
```

[4]RFC 3549

```
#define NETLINK_FIREWALL   3 /* Firewalling hook          */
#define NETLINK_INET_DIAG  4 /* INET socket monitoring */
#define NETLINK_NFLOG      5 /* netfilter/iptables ULOG*/
```

Besides those protocols, it is also feasible to define other protocols for IPC purpose. That is particularly helpful for user-programmed kernel module (like the traffic monitor that designed and described above). In order to do that, the user only needs to define protocol and use this protocol in creating the socket. The *#define* macro does not need to be in netlink.h but can be anywhere of the code as long as it is included. In our program, we have defined the our own protocol *NETLINK_TRAFFICMONITOR*.

After we define the protocol, we can simply create a Netlink socket in the user-space program(our main program for the framework) like what we do in all other sockets by calling the *int socket(int domain, int type, int protocol)* function like this:

```
sock_fd = socket(PF_NETLINK, SOCK_RAW,
  NETLINK_TRAFFICMONITOR);
```

In the kernel program, we must define the interface too:

```
nl_sk = netlink_kernel_create(&init_net,
  NETLINK_TRAFFICMONITOR, 0, processRequest,
  NULL, THIS_MODULE);
```

So far, we have all we need to go through with our traffic monitor and processing.

In the software starting-up phase, the user-space program needs to make the necessary initialization to communicate with kernel using Netlink. Typical steps include opening the socket (using socket () like what we described), creating two socket address structures for Netlink with the type of *sockaddr_nl*, and making a message body of the type *msghdr*. Since the kernel program needs to know the process id(PID) of user-space program to send back the packet information, in the starting-up phase we also make the user-space program sent one empty message body with the PID stored in the socket address structure, so that the kernel program remembers this number in the future use.

After the initialization, the kernel starts to process the TCP packets. Like we mentioned before, all the packets, incoming or outgoing, are processed by the netfilter hook we have registered. The kernel program maintains a table of all the active TCP connections, using the TCP port number as the identifier (since different TCP connections are using different port numbers). When

the TCP hand-shake packets arrives or are sent out, the kernel record the TCP connection status. Possible states include initiating a connection (SYN packet sent, waiting for SYN-ACK), acknowledging a connection(SYN-ACK packet sent, waiting for ACK), and connection established(both ends send normal TCP packets and acknowledge packets). For all of these packets, the kernel collects such information as packet timestamp, packet size, as well as TCP receive window size (for network monitor case). As long as the connection is established, the kernel will send all of the information, except the TCP receive window size, to the user-space program for further processing. Whenever the connection is released, the kernel program deletes this connection record the relevant information from its table.

In the user-space program, after receiving the packets statistics, it can start to calculate the TCP burst information. For example, if the packet interval between two consecutive packets belonging to the same TCP connection is greater than a threshold value(called burst interval threshold), let us say, 10ms, we assume that these two packets belong to different bursts, in another word, the previous burst has ended and there is a start packet of a new burst. If the packet interval described earlier is smaller than burst interval threshold, we assume they belong to the same burst. One thing to note is that in this version of software, the threshold value does not get changed from its default value. We will discuss about the burst interval threshold and verify its rationality in the section 6.3. In a second case, the newly-arrived packet belongs to the existing burst, but it makes the accumulated burst size exceed a specific value(we need a burst size threshold value here), we predict that this is the last packet of one burst(needless to say, we can make wrong judgment). As we talked before, because the burst sizes vary from different TCP servers and shaping algorithms, this burst size threshold needs to get updates whenever we have got a new piece of information of the burst size. We will discuss the threshold update algorithm in the section 6.3 too. What is the difference between these two cases of burst ending? Well, the second one is our prediction of the ending of an old (already existing and counting) burst and the first one is the actual starting of a new burst. In an ideal world, the packet interval between the packet that we just predicted as burst end and the packet next to it shall be greater than the packet threshold, meaning that our prediction is actually correct. Of course there is a third situation: the packet interval is smaller than the burst interval threshold and the accumulated burst size is still smaller than the burst size threshold. In this case we just simply update the accumulated burst size and wait for the next packet.

The reader shall not forget that our prediction could be wrong(in fact, in

the next chapter of the test case analysis the user can see that it is actually pretty hard to keep the prediction accuracy even at a fair rate. But as what we have mentioned in section 1.3 and will discuss in chapter 7, this problem is beyond the scope in this article and needs to be improved in the future). This wrong prediction leads to another situation we should update the burst size threshold (the first situation is when we have the actual size of a newly-ended burst size).

When the software predicts that an burst has ended, we want to take actions to save some extra energy. As we mentioned in the section 3.5, the PSM has already become a common technology to save the energy. It will not be fair if our action against the burst-ended event is just to enable PSM – we have already proved that the PSM can indeed save considerable amount of energy, so it is not a good reference using the system without PSM. But we surely can modify the PSM timeout to some smaller values – the sooner the WNI is in SLEEP mode, the more energy it can save. We can even set the PSM timeout as 0, meaning the WNI will go to SLEEP mode right after the ending packet. But it is not a wise action since our prediction could be wrong, and to wake up the WNI frequently needs to consume more energy.

Like we have introduced in the section 5.2, the PSM timeout setting can be done by calling the *set_WLAN_timeout()* function of the class *WLANAdaptor*.

5.4 TCP receive window size setting

Now we are going to introduce how we set the TCP receive window size because we have already got the necessary background. We have mentioned that the TCP receive window manipulation is one of the main technologies for TCP congestion control. Many algorithms have been designed to fine-tune the window size. However, it is not our purpose to explore these advanced algorithms. We only want to set the receive window size to zero when the SNR drops to a really low level, and restore it to the previous value before set to 0 when the SNR goes back to acceptable range. In order to do that, we need to use our kernel program which is using netfilter to modify the TCP packets on kernel level directly and netlink to send the command from user-space program. Specifically, we are again using the TrafficMonitor class to do this setting.

The user-space program is quite simple: since we have already established the communication with kernel program in the initialization – we have opened a socket to send the message to kernel, and the kernel is aware of the process

ID of user-space program, we can easily send the command to kernel using netlink. The message can be just pre-defined integer or strings.

On the kernel side, our program is keeping a table of all the active TCP connection and some of their properties, among which there is the TCP window size. It also has one common property to indicate whether all the TCP connections should be paused (by setting the TCP windows size to zero) and which of them have already been commanded to pause. So whenever the kernel receives the command to set the windows size to zero, it sets this common property to true. We already know that all the TCP packets, whether downlink or uplink, will go through netfilter's hook. Hence, whenever there is uplink packet belonging to a TCP connection that has not been notified of zero window size is being sent out, netfilter will modify the TCP receiving window size bits to zero to let the far-end server know that the client is not able to receive any packet right now. Similarly, when the kernel receives the command to restore the TCP window, it will set it to the previous values by modifying uplink packets.

How do we determine when we should send the command to modify the TCP windows size(and actually put the WNI into sleep mode after that)? That is the content of our next section: the event processing implementation.

5.5 Event processing and rule definitions

The class SituationProcessor is the core of our event processing system. It is an implementation of what most literatures call processing engine.

At start-up, the SituationProcessor loads the event processing rules that we specify in the main program. The rules define at which circumstance an event is considered to happen, and what adaption we should take against this event. Besides, since we support complex event processing, a complex event which consists of several events and logical relations among these events is also possible to define and process. The rules are defined in format of XML file since its structured feature makes it really feasible to read by users and to process by program. Here is a code snapshot of one rule definition.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<rule xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
      xsi:noNamespaceSchemaLocation="rule.xsd">
  <on>NEW_PACKET</on>
  <if>
    <complex operation="AND">
```

```
        <atomic>
          <capability>pBurstInterval</capability>
          <operation>GREATER</operation>
          <value>TRESHOLDA</value>
        </atomic>
        <atomic>
          <capability>packetSize</capability>
          <operation>SMALLER</operation>
          <value>THRESHOLDB</value>
        </atomic>
        <atomic>
          <capability>currentBurstSize</capability>
          <operation>GREATER</operation>
          <value>TRRESHOLDC</value>
        </atomic>
        <atomic>
          <capability>timestamp</capability>
          <operation>NOTSMALLER</operation>
          <value>beginningOfNextInterval</value>
        </atomic>
      </complex>
    </if>
    <do>
      <type>SET</type>
      <name>Timer</name>
      <param>timer0</param>
      <param>length0</param>
    </do>
</rule>
```

The tag *on* marks which event this adaption is designed to handle – in this example, this adaption is taken when the event *NEW_PACKET* happens (whenever a new TCP packet arrives). The tag *if* means instead of taking this adaption whenever this event happens, it will be taken only when some specific conditions hold, and the children elements describe these conditions. The reader can see we are using the logical operation *AND* to indicate that only when all of these sub conditions hold we take this adaption. Other possible logical operations include *OR, NOT, FOLLOWED. FOLLOWED* means a successive relation. The tag *atomic* means its children elements describe an atomic event, e.g., the atomic element in event processing's world. As we mentioned before, currently an atomic event means a numerical relation

between the capability value and a compared threshold value. All possible operations include *"GREATER", "SMALLER", "NOTGREATER", "NOTSMALLER", "EQUALTO", "NOTEQUALTO"*. In the future we can make it support other relations than numerical comparisons. When these conditions hold, the SituationProcessor will make the adaption defined in the *do* tag. We defined the adaption types *SET, SUB, UNSUBS*. The *name* refers to the target, the first *param* refers to the parameter name of the target, and the second *param* refers to the value we want to set. All the rules are loaded by the function *SituationProcessor::loadRules(const char* filename)* during the initialization.

After the initialization, the SituationProcessor generates the sensors according to the rule and event definitions, starts each sensor in its sensor list, and each sensor then begins their monitoring at defined frequency. It is these sensors' responsibilities to generate the atomic and complex events. Whenever a new event is generated, either atomic or complex, the sensor inserts it to the event queue of the SituationProcessor.

The purpose of *SituationProcessor::run()* is to process the events inserted to its event queue. It needs to be ran at another thread than the main thread. We will introduce multithreading in the section 5.6.

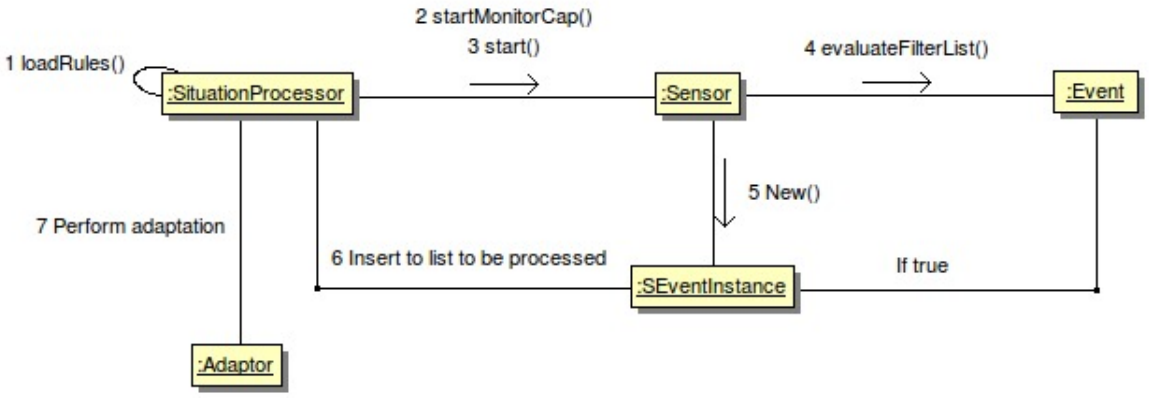


Figure 5.4: Event subscription

Now let us see how the event is subscribed and processed. Figure 5.4 is a UML communication diagram which shows how it works. In the beginning the SituationProcessor loads the rules written in XML file, as we described above. It will then register each interested capability to the corresponding

Sensor, and make the Sensor start to monitor such capability. Each Sensor has a list of Event, which again is defined in the XML file. The Sensor is evaluating against the conditions of every event in this list all the time. Whenever an Event is evaluated to be happened, the Sensor will generate one SEventInstance object, which has the information of adaptations(our rules in the XML file), and insert it to the list for all events to be processed. There is only one common list for all the Sensor, and this list is accessible to SituationProcessor. The SituationProcessor is processing all events in this list in another thread. When processing each event, it will make Adaptor to perform the corresponding adaptations.

5.6 Multithreading

In this section we want to discuss about the importance of multithreading and how we do that in our software.

Firstly maybe we should talk about the difference between a process and a thread. A simplified description of a process is a whole program running on the operating system (of course, multi-processing controlled by a main function is also possible. Another case is like in our software, two processes – the kernel and the user-space programs – are running individually. But in both cases can be treated as different programs in the narrow sense). On the operating system level, when a process is running, the OS firstly assign the necessary resources to it: the entry and exit points of the program, the memory space, and of course the process ID on Linux systems, so that it can be treated as the smallest unit from the OS's point of view. Different processes will have totally different memory space, and in the multi-process programming, whenever a new process is started (in Linux, typically using *pid_t fork(void)* function), it will apply for a new bulk of memory and copy all the current variables into it. After that, the two processes will have their own copy of the variables.

Unlike a process, a thread is running inside a process, and all of the threads running on the same process share the only one copy of the memory and variables. Each individual thread does not own all the necessary system resources, instead, only some stacks and registers are unique for each of them. The point of using multi-thread is that some tasks can be run simultaneously while still do not use too much system resources. Besides, in multi-processing, it requires advanced inter-processing communications technologies to for example, make some variables consistent with the copies in other processes,

but since the threads share only one set of variables, it is easy for them to communicate with each other. On the other hand, the coin has two sides, in multi-threading it brings the problem that sometimes more than one threads will be trying to change the value of the same variable. There needs to have some mechanism to ensure that only one thread can modify a variable at one time. We will talk about it later. Another difference is, since the processes have their own memory spaces, whenever a process is crashed down, other ones will not get effected if they are ran under protection mode. However, since all threads under one process share the unique memory space, when one thread crashes, it might make all other threads stop. So the software designer needs to pay special attention to this too. But this is not our main target in this project, so we did not put too much strength on that. To summarize, the operating system sees different processes as different units that it is supposed to schedule and allocate the resources, while it sees the threads running in one process as a whole unit.

Now we are going to see how we implement the multi-threading in our software. As mentioned before, the functions *run()* of each sensor and *SituationProcessor::run()* are running in a separate threads.

The reasons to run different sensors on different threads are: 1. The amount of sensors are not huge so that the cost of multi-threading is acceptable. Currently we only have two specific sensors, but even in the future when more sensors are designed, we can predict that there will be tens of sensors at most. Of course the number of capabilities to monitor could be more than one hundred. Luckily, a sensor can monitor more than one capability. 2. Each individual sensor is in charge of a unique set of capabilities monitoring, and these sets are most likely irrelevant to each other. These give us a desired situation to deploy multi-threading to make our software more efficient. We can imagine that if we do not use multi-threading technology, instead, we query all of the capabilities in one thread, then we must design a monitoring scheduler which will query all these capabilities at a sequence of time. To make it even worse, the capabilities need to be queried at different frequencies. The scheduling algorithm would be complex and yet the performance would not be good. But by using multi-threading, a sensor can query one set of similar capabilities at a set frequency, while other sensors do their tasks at the same time under different frequencies.

Glib [5] is a cross-platform utility library which produces many use functionality. It was firstly part of GTK+, which aimed to provide a UI design framework. But the non-UI part was separately released as Glib later. An useful

[5] http://developer.gnome.org/glib/stable/

function to create a new thread is a static function in the Glib::Thread class: *static Thread* create(const sigc::slot <void>&slot, bool joinable)*, which creates a thread that runs the function pointed by slot, specified by the user whether it is joinable, and with the default priority of *THREAD_PRIORITY _NORMAL*. For example, for *NetworkMonitor*, in order to use this function, we use such code inside the *start()* function of each sensor:

```
try
{
  Glib::Thread::create(sigc::mem_fun(*this,
    &NetworkMonitor::run), false);
}

catch(Glib::Thread::Exit&)
{
  std::cout<<__PRETTY_FUNCTION__<<"::NetworkMonitor
    thread exits"<<std::endl;
}
```

Inside the try block we create a thread, which will run the *NetworkMonitor::run*. Whenever the thread is ended either by the situation processor or when it ends after a set period, the catch block will do the ending job. In current work, it only outputs the result to the user. Needless to say, the *NetworkMonitor::run* function does the actual network monitoring work(to monitor the SNR value, for instance).

For *SituationProcessor::run()*, we have mentioned in section 5.5 that it needs at least two threads: one is the main thread that our main program is running on, the other one is in charge of loading and processing the events, which are generated and inserted by the sensors, in its event queue. The function that does the loading and processing is called *run()*, so similarly, we will create the thread like this:

```
Glib::Thread::create(sigc::mem_fun
  (*this, &SituationProcessor::run), false);
```

The event queue (implemented as a standard C++ list) is a member of situation processor, and it is accessible to all the sensors. Whenever a sensor generates an event, it will insert it into the queue. So we will have to face the fact that sometimes more than one sensors will need to insert to the queue at the same time. If we cannot make good management, it will easily crash the thole thread(and as we talked, as a result the whole process) by letting two pieces of code touch the same memory at the same time. In order to

coordinate it, we use a so-called mutex technology.

A mutex is a common variable to all the threads, and it has basically two status: locking and unlocked. At one time, any and only one thread can lock it, and before this thread unlocks it, all other attempts to lock the mutex will be blocked. By using the mutex, we can make the thread to modify the queue only when itself is locking the mutex, and when it finishes the operation to the queue, it will unlock it to let other threads, or maybe itself again, operate on it.

In Glib, there are two data structures that we use to implement the mutex: *GMutex* and*GCond*. The *GMutex* works exactly like we described, in order to lock the mutex (say it is called data_mutex), we called *g_mutex_unlock(data _mutex);* and to unlock it, we call *g_mutex_unlock(data_mutex);* so we only need to say something about *GCond*. The pseudo code that we manipulate the mutex is like this:

In the *SituationProcessor::run()*, it runs this constantly:

```
g_mutex_lock(data_mutex);
while (eventQueue->size() == 0)
    g_cond_wait(data_cond, data_mutex);

POP UP ONE EVENT
SEND THE EVENT TO EVENT HANDLE TO PROCESS

g_mutex_unlock(data_mutex);
```

After the situation processor gets the mutex, it firstly checks whether there is any event in the queue. If the queue is empty, it will actually give up the mutex (although it does not call *g_mutex_unlock()* explicitly), and wait until the condition *data_cond* to be true. After that, it will try to get the lock (again, without explicitly calling *g_mutex_lock()*). The condition will only be true and sent by the sensor's code whenever it successfully inserts an event. As long as the situation processor gets the lock after this condition holds, the code continues, and it will actually pop up one event and send it to the event handler to process. Correspondingly, the pseudo code in the sensor part is:

```
g_mutex_lock(data_mutex);
INSERT ONE EVENT
g_cond_signal(data_cond);
g_mutex_unlock(data_mutex);
```

Just like we mentioned, the sensor sets the condition by calling the function *g_cond_signal(data_cond)*. By using the *GMutex* and *GCond* together, we

have managed the event queue insertion and popping up by different threads.

There is some other issue about multi-threading, for example, to prevent the deadlock of the mutex. But we are not going to dig them more.

So far, we have introduced enough details for the reader to understand what are essential in our work and how we implement them. In the next chapter, we will see some test cases and check the results, some discusses upon which will be given too.

Chapter 6

Evaluation

In this chapter we will test our resource management framework in two power management scenarios. The first scenario takes adaptation based on SNR measurement. It is aiming to deal with environment and link quality changes. The second scenario is based on TCP traffic content, meaning we make adjustment by analyzing the TCP packets.

6.1 General terms

6.1.1 SNR prediction algorithms

There are already several prediction algorithms such as Auto-regressive Integrated Moving Average (ARIMA) [19]. But ARIMA requires four parameters and linear calculation. Specifically, it needs to be trained to get such parameters before this model can be applied, and the values of these parameters should be re-calculated when the channel model changes.

As stated in 1.3, we are not aiming to design a delicate SNR prediction algorithm. In our test we use a pretty simple prediction algorithm: we use the previous monitored SNR value as the prediction of the value in next time. We use such algorithm based on two observations: If the moving speed is low enough and if the SNR monitor frequency is relatively high (these two conditions guarantee that the link quality does not change so fast that our prediction algorithm cannot follow the changes), our simple prediction algorithm can produce a good result.

Event Name	Event Description
LOW_TO_HIGH_SNR	When the last monitored SNR value is below the threshold and next predicted SNR is above the threshold
HIGH_TO_LOW_SNR	When the last monitored SNR value is above the threshold and next predicted SNR is below the threshold
NEW_PACKET	When there is a downlink TCP packet coming or a uplink TCP packet going out of the client
NEW_BURST	We predict that a packet is the first packet of a new TCP burst
END_BURST	We predict that the last TCP packet we analyzed is the last one of current TCP burst
WRONG_END_BURST	We predicted the END_BURST event, but that packet turns out not to be the last packet of a burst

Table 6.1: Definitions of all used events

6.1.2 All event definitions

In our test we have defined multiple events. As we mentioned, we test our framework in two power management scenarios, so most of the events are about system resources that related to power management. Table 6.1 shows the definitions of all those events with some descriptions. The first two events are used in scenario 1, and the rest four events are used in scenario 2. More details can be found in 6.2 and 6.3.

6.2 Scenario 1: Transmission adaptation using based on SNR measurement

6.2.1 Use case

The use case of Scenario 1 is as follows: Alice is downloading a video clap from a TCP server (called TS in the following content) to her N900 mobile phone while she is walking around inside a building. Both of the mobile phone and the TCP server are connected to the same Wi-Fi access point (AP) and hence all the TCP traffic is routed through this AP.

Since Alice is walking all the time during the whole transmission period, and such surrounding objects as walls and doors are attenuating or even blocking the wireless signal, it is not hard to understand that there will be some amount of energy waste when the transmission condition is bad. For example, the phone will have to increase the transmission power to make sure that the request and acknowledge packets are sent to the AP successfully. When the transmission condition is even worse, the phone will end up keeping sending those packets and never getting any packet from the AP. We see big space here to take our adaptations to the mobile phone resource under these conditions to avoid such kind of waste: when the transmission condition is really bad, it is wise to just shut down the TCP transmission and resume it when the condition goes back to good. As [25] has already pointed out, the value of signal-to-noise-ratio (SNR) can be considered as a good measurement to the link quality. It is also applicable to our case since SNR is changing corresponding to the movement and environment change. As a result, we are using SNR in our software as the criteria against which adaptations are taken.

6.2.2 Settings

Figure 6.1 shows the environment objects and the moving pattern of Alice. The point where she starts to walk and to download the file is A, which is the closed position to the wireless router, i.e., the AP. There is a thin wall between A and AP. Alice first walks towards B, and then she turns left and goes towards C. At point C she waits for about 15 seconds, after that she goes back to B, and then to A. Note that the signal attenuation on the way B<=>C is higher than the one on the way A<=>B, that is because there is another wall which blocks the signal. Alice keeps repeating such moving route until the transmission finishes (we can assume that she is walking around thinking about the business plan she has been working with). She walks in a constant speed of 0.4m/s (again, she is thinking about something so she walks slowly). The typical time usage is, A to B 20s, B to C 10s, waiting at C 15s, and some other time such as turning around.

The software is monitoring the SNR value and the sampling rate is 1Hz. We set a threshold value of 15dB for SNR. A value lower than 15dB is considered as low SNR condition and a value above it as high SNR. We define two events in this scenario: LOW_TO_HIGH_SNR and HIGH_TO_LOW_SNR. When our software observes a low SNR value and predicts that the value of the next time slot will be a high SNR value, it generates a LOW_TO_HIGH_SNR event. Similarly HIGH_TO_LOW_SNR event is defined as the monitored

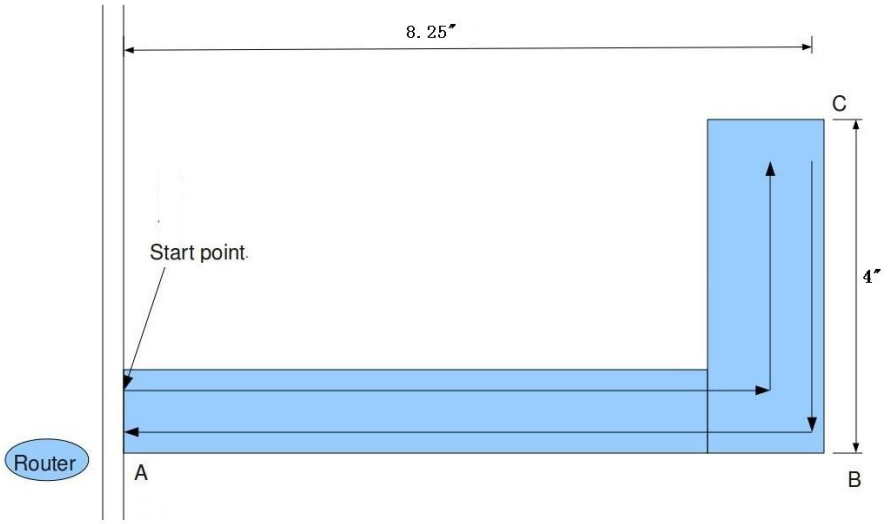


Figure 6.1: Environment and moving pattern

value is high and prediction value is low. We set two rules with respect to these two events.

Rule 1: when HIGH_TO_LOW_SNR event happens, which means the signal condition becomes bad, the software will pause the TCP transmission by setting the receive window size to 0, the previous window size will be saved for later use. After setting the windows size, the WNI is put into SLEEP mode.

Rule 2: when LOW_TO_HIGH_SNR event happens, which means the signal condition is going to be good again, the software will resume the TCP transmission by restoring the receive window size to previous value. After restoring the window size, we put WNI into IDLE mode.

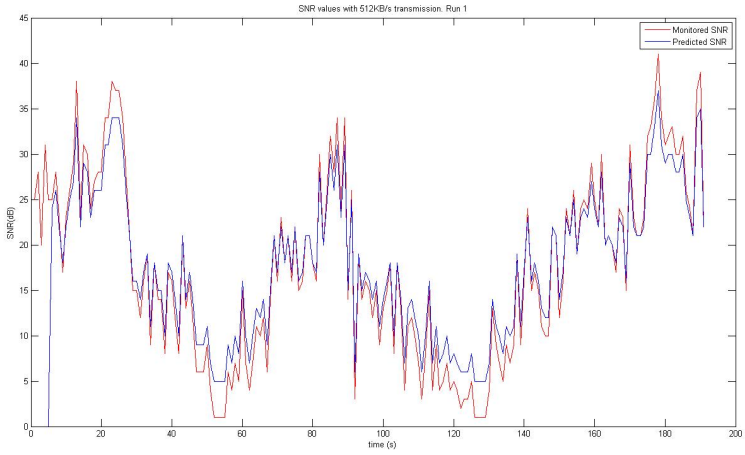
The size of the file to download is 39.3MB, and the maximum download speed is limited to 512KB/s on the server side using Trickle[1]. We first download the file without our software to take the result as reference. Then we run SNR monitor and event processing but without adaptation, i.e, when the even happens, the event engine just simply does nothing. The purpose of this test is to get the overhead of our software. Finally, we run it without adaptation to see whether we can save some energy or not. All the same tests are ran 3 times. We run Wireshark[2] on both mobile phone and TCP

[1]http://monkey.org/ marius/pages/?page=trickle
[2]http://www.wireshark.org/

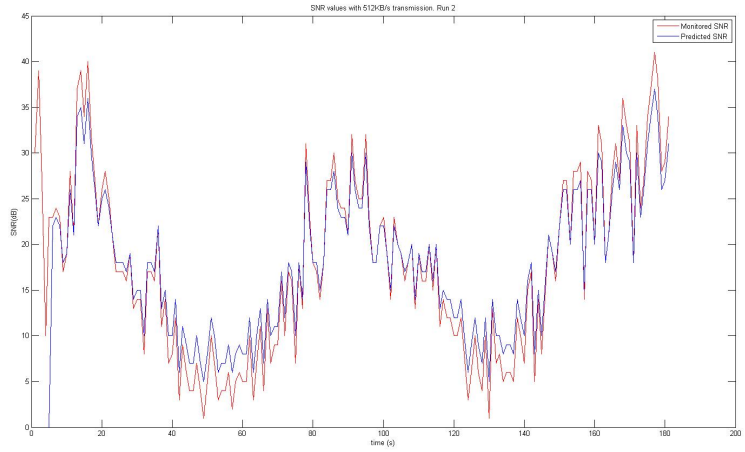
server to monitor the TCP packets transmission and measure for example, the throughput and transmission pause duration.

6.2.3 Analysis

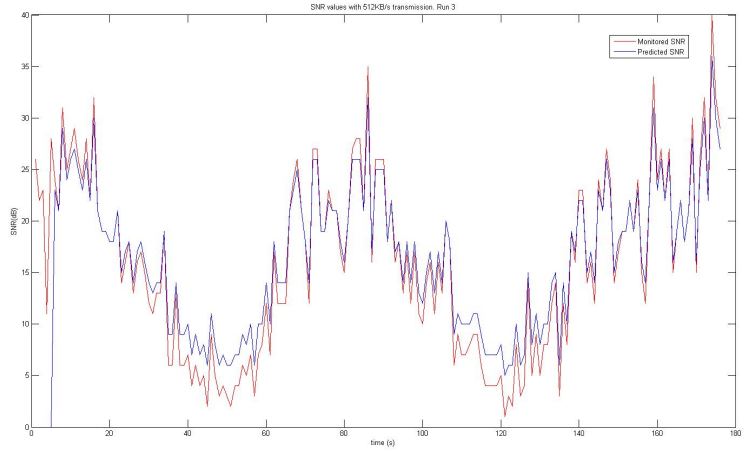
The first thing we need to discuss is the prediction accuracy. After all, all the further actions are taken on the base of the prediction, and if the prediction algorithm cannot produce a good estimation of SNR value with acceptable error, we will not get a satisfying result. As mentioned on section 6.1.1, several algorithms have already been proposed but they all have their drawbacks. The main problem is, most of these algorithms are somehow static, which require offline training before these models are being used in real life, and if the network condition changes dramatically, a new training is needed. For example, ARIMA [19] has as simple as four parameters and linear calculation. However, values of these parameters need to be calculated beforehand, and if the moving pattern or the surrounding objects get changed, these values need to be re-calculated. We are going to prove that if the walking speed is as low as our test, and if the SNR monitor frequency is high enough, our simple prediction algorithm described in section 6.1.1 can produce a good result.



(a) run1



(b) run2



(c) run3

Figure 6.2: SNR Comparison between prediction and actual value

Figure 6.2 shows the comparison between the prediction and the actual SNR values. We can get the impression that our simple prediction algorithm is good enough to follow the change of SNR values. A quantitative measurement to the accuracy is to check the mean squared error (MSE), which is the average of squared error of the estimation. In our test, the average MSE among all the experiments is 34.87, which is pretty good compared to for example [19].

Next we shall check the SNR threshold value. Figure 6.3 shows the cumulative distribution function (CDF) of SNR monitored values. We can see that the SNR values are uniformly distributed with a mean value of 17dB and standard deviation of 9dB. So our choice of 15 as the threshold value is reasonable.

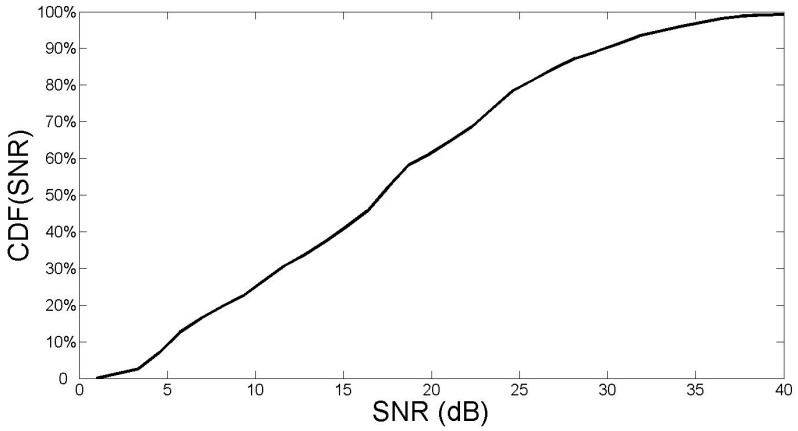


Figure 6.3: CDF of SNR values

Figure 6.4 shows one typical set throughputs between the experiments with and without adaptations. Some observations from this figure:

1. The duration of the whole download without adaptation is 161.97s, whereas without adaptation is 229.15s. There is 41.5% more time consumption. It is understandable because under low SNR values we turn WNI into SLEEP mode, however, there might still some packet transmission if we do not do this (without adaptation case) with the cost of more power consumption. The extra power consumption comes from for example, packet re-transmission. So the total transmission time will be longer with our adaptation, but our main target is to save the power, and we must trade-off between saving power and having longer transmission duration.

2. In the experiment without adaptation, the transmission was paused for 29.80s. This happens when the SNR drops at a really low value so that the packet transmission becomes impossible. On the contrary, with adaptation, the duration of the pause due to our action to put the WNI into SLEEP mode is 122.04s. In another word, the actual duration of the transmission when the WNI is in RECEIVE mode using adaptation is 229.15-122.04=107.11s, while this value in the case without adaptation is 161.97-29.80=132.17s. The actual transmission duration, during which the WNI consumes most of the power, is shorter using our adaptation than not using it. Now let us take all three experiments for each test into account. Averagely, the actual transmission time without adaptations is 140.30s, while with adaptations is 109.21s. As a result, with adaptations, the actual transmission time saving is about (140.30-109.21)/140.30=22.2%. This is also not a surprise because the WNI consumes less power when the link quality is good than when it is bad to transmit the same amount of packets.

3. Because there is no software to monitor the power consumption while the mobile phone is moving (we can use such software when the phone is stable in *Scenario 2*), we turn to the formula described in [30] and [13]. We can see that we reduced the power consumption from 92.15J to 82.30J to transfer the same amount of data, i.e., we saved about 10.69% energy using our framework and event-action definition. The table 6.2 shows the detailed results.

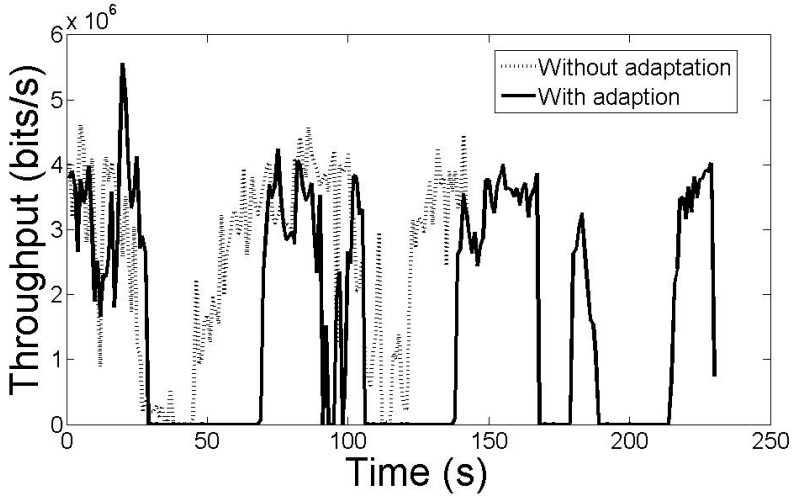


Figure 6.4: Network throughput comparison between the test with and without adaptation

	Without adaptation	**With adaptation**
Total transmission time	171.41S	233.60S
SLEEP mode duration	31.11S	124.39S
RECEIVE or IDLE mode duration	140.30S	109.21S
Power consumption	92.15J	82.30J

Table 6.2: Power and time consumptions in scenario 1

6.3 Scenario 2: Transmission adaptation based on traffic content

6.3.1 Use case

The use case of scenario 2 is like this: Alice is listening to an online radio program called **SuomiPop**[3], which is using TCP to transmit, on her N900 phone. The phone is still connected to the AP we used in scenario 1. This time she is not moving so we can assume the link quality is good and stable.

[3]http://217.30.180.242:8000/spo.mp3

We know that there is a certain kind of pattern for TCP traffic, especially online streaming traffic. For example, the burst size (a TCP burst is defined as a sequence of TCP packets, which arrive close to each other, and there is a relatively long period between two TCP burst are sent), burst interval (the length of the period from the ending of one burst to the beginning of next burst) all have a specific pattern for TCP streaming. This kind of pattern comes from the function of shaping from the streaming server. If the burst interval is big enough, we can put the WNI into SLEEP mode during such interval to save some energy. This is exactly what we are doing. Of course, if our prediction is wrong we might end up with some inefficiency. For example, if we predict that a burst has ended and take our action, but actually it is not finished and there are some continuing packets right after the WNI is set to SLEEP mode, the WNI will be woke up hence some delay and energy waste are brought here. We are going to discuss about the effects of errors in the Analysis part.

6.3.2 Settings

We used a Monsoon power monitor[4] to act as the DC power supply and at the same time to monitor the power consumption. A Windows application program is available to configure and control the power monitor. We set the sampling frequency of power monitor as 1MHz, which means the power monitor measures the values of current, voltage, power and so on 1000000 times a second. This program is also used to calculate the total power consumption and average power during transmission.

The PSM is enabled without adaptation to provide a good reference. The timeout value is set to 200ms. The PSM timeout works like this: when there is no traffic during such period, the WNI is put into SLEEP mode. The main action in our adaptations is also to change this timeout, making it shorter so that the WNI is put into SLEEP mode more frequently.

We set the packet interval threshold as 10ms, i.e., if the interval between two consecutive packets is smaller than 10ms, we consider these two packets belonging to the same burst.

The burst size threshold is initialized as 4000 bytes, which means if a packet make the cumulative size of a burst exceed this value, we predict that this is the last packet of the burst. This threshold value gets updated during run-time. Specifically, it is updated as the average value of the last five burst

[4]http://www.msoon.com/LabEquipment/PowerMonitor/

sizes. But if a wrong prediction about burst ending happens, we just simply double the threshold so that next time this kind of wrong prediction is less likely to happen.

The beacon value in the AP is set as 100ms, which means every 100ms, the AP is sending a special beacon packet to all the devices connected to itself. Inside the beacon it informs about the existance of targeting traffic. When a device wakes up (the interval may be different, usually larger than the beacon interval) it checks whether there are incoming packets or not. We test the program for 4 minutes. There are 4 events involved in this test, namely, NEW_PACKET, NEW_BURST, END_BURST, and WRONG_END_BURST. The rules are defined as follows:

Rule 1: when NEW_PACKET event happens, as the names indicates, a new packet has arrived. We analyze this packet. Specifically, if the packet interval between the arriving packet and the previous packet is greater than current packet interval threshold, we generate the event NEW_BURST; if the packet interval is smaller than current packet interval threshold **AND** the current burst size is greater than current burst size threshold, we generate the event END_BURST; if the packet interval is smaller than current packet interval threshold **AND** the current burst size is smaller than current burst size threshold, we update current burst size.

Rule 2: when NEW_BURST event happens, which means there is a first packet of a new burst coming, we put the PSM timeout value as 100ms, which is long enough for next packet.

Rule 3: when END_BURST event happens, which means we predict that a final packet of one burst has arrived, we set the PSM timeout value as short as 5ms, waiting for potential incoming packet, if there is no packet in this 5ms, the WNI is put into SLEEP mode. We also update the current burst size threshold after we have received a new whole burst.

Rule 4: when WRONG_END_BURST event happens, which means we made a wrong predict that about burst ending but in fact it has not finished, we should double the current burst size threshold so that next time there is less chance that we make this kind of wrong prediction.

6.3.3 Analysis

As mentioned before, the mobile phone's power consumption in different modes can vary quite distinctly. We first calculate the power value for these modes. Table 6.3 shows the power value in IDLE, SLEEP and RECEIVE

IDLE	SLEEP	RECEIVE
668.88mW	32.25mW	980.68mW

Table 6.3: Power consumption of N900 when WNI is in different modes

mode. The phone in SLEEP mode consumes quite low energy, only 4.8% as in IDLE mode and 3.3% as in RECEIVE mode. On the contrary, when the WNI is in IDLE mode, even if there is no data transmission, it still consumes 68.2% of the energy as when there is actual data transmission. It proves our idea that if we can put WNI into SLEEP mode correctly, instead of letting it stay at IDLE mode, when there is no packet transmission, we will save much energy compared to letting it stay in IDLE mode.

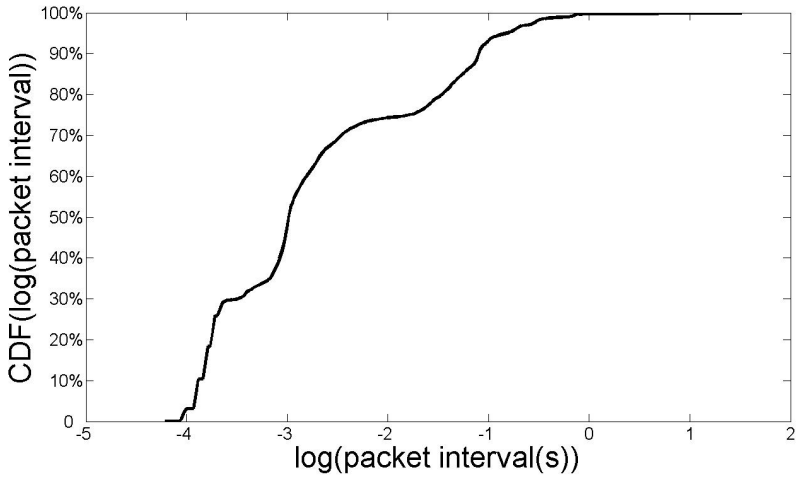


Figure 6.5: CDF of packet interval of an online radio program

Next we analyze the packet interval threshold. Figure 6.5 shows the CDF of logarithm value of packet interval. We can see that 72.6% of the packets have a packet interval smaller than 10ms, which means our threshold is fairly set.

Figure 6.6 shows the CDF of logarithm value of burst size. It indicates that almost 68.5% of the bursts have the burst size greater than 4000 bytes. So the initial burst size threshold is also set reasonably. Of course, we are also updating this threshold during run-time, which makes it more content-specific.

Table 6.4 shows the actual power values with and without adaptation. We can see that with our adaptation, it consumes a little bit more energy (about

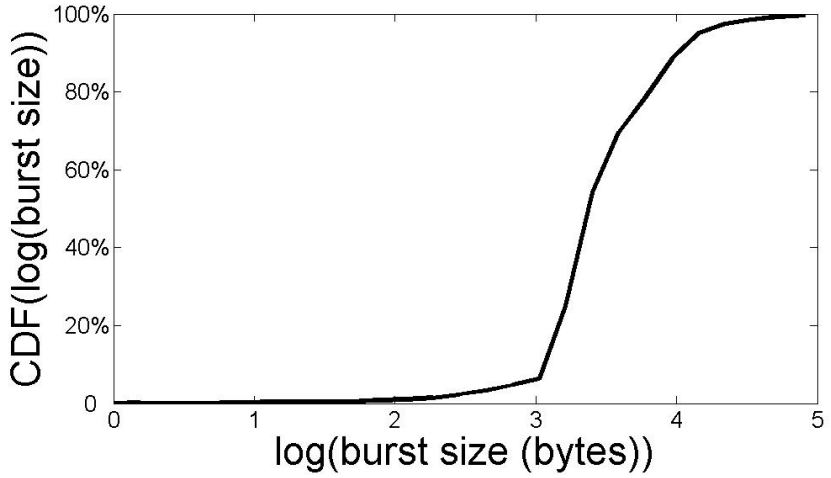


Figure 6.6: CDF of burst size of an online radio program

Without adaptation	980.75mW
With adaptation	1014.68mW

Table 6.4: Power consumption with and without running our software

3.5% more) than without it. There might be two reasons causing it.

The first reason is, our burst prediction algorithm might not be able to provide a good estimation. There are two kinds of estimation error in our algorithm. **1.** If our software predicts that a burst has ended(we call it false positive) and puts WNI into SLEEP mode, but in fact there is still some packets belonging to the current burst, the WNI will be woke up right after it is in SLEEP mode. This kind of behavior does not save energy, on the contrary, it may cause some energy waste since putting into SLEEP and waking up also uses some energy. **2.** If there is indeed a burst end packet arrived but our software could not predict it(we call it false negative). The WNI will just stay in IDLE mode for a little bit longer until the PSM timeout and then in SLEEP mode. This kind of behavior does not cause energy waste, but it makes our software less efficient. From table 6.5 we can see that the second kind of error happens about 50.27%, while the first kind of error happens about 33.42%. The high rate of estimation is the main reason we cannot get a satisfying result. The packet pattern prediction without any knowledge of the content beforehand is not an easy work to do, we need to think about how to make a better prediction algorithm.

False positive	33.42%
False negative	50.27%
Correct prediction	16.31%

Table 6.5: Power consumption with and without running our software

Secondly, without adaptation, the PSM is still enabled. If there is no packet coming in the timeout period (200ms in our experiment), the WNI will be put into SLEEP mode. It means that the WNI might still be able to stay in SLEEP mode for a certain period without any adaptation. As a result, although we noticed the huge difference in the power consumption between IDLE mode and SLEEP mode, in real life with PSM enabled, it seldom happens for the WNI to stay in IDLE mode for a long time.

In this chapter we evaluated our framework using two power management scenarios. We proved that with proper designed event-action, and reasonable chosen parameters, we can save the energy consumption in some usage cases.

Chapter 7

Discussion

In this chapter we are going to talk about some future work needs to be done to improve the performance of our software.

Firstly of all, our current event processing engine is still quite simple: whenever an event is generated, it will be inserted to a common event queue, where all the events are here to be processed following a first-in-first-serve pattern. However, it is possible that some events might be more important than others, so that they need to be processed sooner than others, even if they happened later than those ones. Hence we think we can implement a priority queue, where events with different levels of importance can be served according to their priorities.

Secondly, we can add more complex logic calculations. Currently we only support *AND* and *OR* calculations. In the future, we can implement the time relation such as *FOLLOWS*. Because this is the basic to describe a time-sequenced relation. With such more complex logic relations, we can design more complicated events that handle more complicated situations.

Thirdly, we will improve our rules for traffic monitor case to get better performance. As we mentioned in 6.3.3, we can improve our prediction algorithm of the burst pattern. Because the accuracy of current algorithm can still be improved.

There are some other things that can be done in the future. For example, we can use more complicated SNR prediction algorithm, and we will design some usage scenarios in other fields than power management.

Chapter 8

Conclusions

In this thesis we have proposed an event-driven framework for mobile system resource management. We talked about the basic ideas of resource management and event processing. We adopted Publish/Subscribe and event processing architectures into our work, and built the solution on Maemo platform. Our solution starts by defining the events to be monitored and the adaptations to take against these events. After the loading, the event processing engine registers to (subscribes to) all events, and makes every sensor monitor the corresponding measurements. Whenever an event happens, the corresponding sensor generates an event and inserts it to the queue to be processed. The event processing engine checks the event status and commands to take adaptation which we define in XML file. By doing it, we provide an solution to using event processing in the resource management.

In order to test our program we have designed two test scenarios: the *adaptation based on SNR measurement* and the *adaptation based on traffic content*. In the first case, we take SNR values as the measurement against which we take our adaptation. But taking such event-adaptation rule, we have made saved the energy consumption by 10.69%, which proves efficiency of our work.

In the second case the adaptations are based on the TCP traffic contents. The sensor is monitoring every TCP packet going through the WNI. We predict the burst pattern and make adaptation based on the predictions. In this case, we observes that it actually consumes a little bit more energy (about 3.5%) than without adaptation. The possible reasons could be that the current PSM settings are already pretty good to handle burst transmission, and we might need to design more delicate algorithms to get improvement.

To make a summary, in our work we have designed a resource management solution based on event-driven architecture. We have also designed some

usage scenarios to prove that it is actually working for on mobile platform.

Bibliography

[1] Ieee standard for information technology - telecommunications and information exchange between systems - local and metropolitan area networks-specific requirements - part 11: Wireless lan medium access control (mac) and physical layer (phy) specifications, June 2007. IEEE Std 802.11-2007 (Revision of IEEE Std 802.11-1999).

[2] Aguilera, M., Strom, R., Sturman, D., Astley, M., and Chandra, T. Matching events in a content-based subscription system. In *Proceedings of the eighteenth annual ACM symposium on Principles of distributed computing* (1999), ACM, pp. 53–61.

[3] Ambler, S. *The elements of UML 2.0 style*. Cambridge Univ Pr, 2005.

[4] Anand, M., Nightingale, E., and Flinn, J. Self-tuning wireless network power management. *Wireless Networks 11*, 4 (2005), 451–469.

[5] Ashwini, H., Thawani, A., and Srikant, Y. Middleware for efficient power management in mobile devices. In *Proceedings of the 3rd international conference on Mobile technology, applications & systems* (2006), ACM, pp. 49–es.

[6] Bailey, J., Poulovassilis, A., and Wood, P. An event-condition-action language for xml. In *Proceedings of the 11th international conference on World Wide Web* (2002), ACM, pp. 486–495.

[7] Baldauf, M., Dustdar, S., and Rosenberg, F. A survey on context-aware systems. *International Journal of Ad Hoc and Ubiquitous Computing 2*, 4 (2007), 263–277.

[8] Bellavista, P., Corradi, A., Montanari, R., and Stefanelli, C. Context-aware middleware for resource management in the wireless internet. *IEEE Transactions on Software Engineering* (2003), 1086–1099.

[9] Choi, S., and Handley, M. Designing tcp-friendly window-based congestion control for real-time multimedia applications. *Inproceedings of PFLDNeT* (2009).

[10] Cugola, G., Jacobsen, H., et al. Using publish/subscribe middleware for mobile systems. *ACM SIGMOBILE Mobile Computing and Communications Review 6*, 4 (2002), 25–33.

[11] Etzion, O., and Niblett, P. *Event Processing in Action*. Manning Publications Co., 2010.

[12] Eugster, P., Felber, P., Guerraoui, R., and Kermarrec, A. The many faces of publish/subscribe. *ACM Computing Surveys (CSUR) 35*, 2 (2003), 114–131.

[13] Friedman, R., Kogan, A., and Krivolapov, Y. On power and throughput tradeoffs of wifi and bluetooth in smartphones. In *INFOCOM, 2011 Proceedings IEEE* (april 2011), pp. 900 –908.

[14] Gamma, E., Helm, R., Johnson, R., Vlissides, J., et al. *Design patterns*, vol. 1. Addison-Wesley Reading, MA, 2002.

[15] Goldsmith, A. *Wireless communications*. Cambridge Univ Pr, 2005.

[16] Harris, C. *Context-Aware Power Management*. PhD thesis, University of Dublin, Trinity College, 2006.

[17] Harris, C., and Cahill, V. An empirical study of the potential for context-aware power management. *UbiComp 2007: Ubiquitous Computing* (2007), 235–252.

[18] Huang, Y., and Garcia-Molina, H. Publish/subscribe in a mobile environment. *Wireless Networks 10*, 6 (2004), 643–652.

[19] Kalyanaraman, S., Xiao, Y., and Ylä-Jääski, A. Network Prediction for Energy-aware Transmission in Mobile Applicatioins. *International Journal on Advances in Telecommunications 3* (2010), 72–82.

[20] Krashinsky, R., and Balakrishnan, H. Minimizing energy for wireless web access with bounded slowdown. In *Proceedings of the 8th annual international conference on Mobile computing and networking* (2002), ACM, pp. 119–130.

[21] Luckham, D. *The power of events: an introduction to complex event processing in distributed enterprise systems*. Addison-Wesley Longman Publishing Co., Inc., 2001.

[22] Michelson, B. Event-driven architecture overview. *Patricia Seybold Group* (2006).

[23] Mo, J., and Walrand, J. Fair end-to-end window-based congestion control. *IEEE/ACM Transactions on Networking (ToN) 8*, 5 (2000), 556–567.

[24] Muhl, G., Fiege, L., and Pietzuch, P. *Distributed event-based systems*. Springer-Verlag, 2006.

[25] Schulman, A., Navda, V., Ramjee, R., Spring, N., Deshpande, P., Grunewald, C., Jain, K., and Padmanabhan, V. Bartendr: a practical approach to energy-aware cellular data scheduling. In *Proceedings of the sixteenth annual international conference on Mobile computing and networking* (2010), ACM, pp. 85–96.

[26] Shih, E., Bahl, P., and Sinclair, M. Wake on wireless: an event driven energy saving strategy for battery operated devices. In *Proceedings of the 8th annual international conference on Mobile computing and networking* (2002), ACM, pp. 160–171.

[27] Tarkoma, S. *Mobile middleware: architecture, patterns and practice*. John Wiley & Sons Inc, 2009.

[28] Weissel, A., and Bellosa, F. Process cruise control: event-driven clock scaling for dynamic power management. In *Proceedings of the 2002 international conference on Compilers, architecture, and synthesis for embedded systems* (2002), ACM, pp. 238–246.

[29] Wood, A., Stankovic, J., Virone, G., Selavo, L., He, Z., Cao, Q., Doan, T., Wu, Y., Fang, L., and Stoleru, R. Context-aware wireless sensor networks for assisted living and residential monitoring. *Network, IEEE 22*, 4 (2008), 26–33.

[30] Xiao, Y., Savolainen, P., Karppanen, A., Siekkinen, M., and Ylä-Jääski, A. Practical power modeling of data transmission over 802.11 g for wireless applications. In *Proceedings of the 1st International Conference on Energy-Efficient Computing and Networking* (2010), ACM, pp. 75–84.

[31] Zheng, R., and Kravets, R. On-demand power management for ad hoc networks. In *INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies* (2003), vol. 1, IEEE, pp. 481–491.