POLITECNICO DI MILANO

Facoltà di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria Informatica

Self-Adaptive Software Systems on Android based on Application Heartbeats Framework

Supervisor: Prof. Marco Domenico Santambrogio

The master thesis of:

Pham Tien Thanh

Matricola n. 736554

Academic Year 2010–2011

Summary

Computing systems have evolved through an unimaginable step in comparision to their first ancient ones. The first electronic computers were developed in the mid-20th century (1940-1945). Originally, they were the size of a large room, consuming as much power as several hundred households and provied very limited computing ability. Our desktop computers today are equipped with multi-core processors, each of each may go up to 6 cores (e.g., Intel Core i7 series and AMD Phenom X6 series), smaller size than ever, consume less power and provide thousand times in computing ability more than the old age ones. Even our mobile devices are now powered by extremely powerful SoC which refer to the ability of integrating all components of a computer or other electronic system into a single integrated circuit(chip). Computing systems which now can be defined as an aggregate of complex computer architectures and software systems, therefore implies their great complexity . Moreover, the born of Internet makes the overall picture becomes more complicated by providing the ability of connecting computers and computer systems to create distributed computing systems. Thus, software systems are also becoming more sophisticated due to the requirements of dealing with extremely powerful computing

systems working in dynamic environments.

Software systems which have to deal with the changing in environment normally require human suppervisors to continue operation in all conditions. These (re-)configuring, troubleshooting, and in general maintaining tasks lead to costly and time-consuming procedures during the operating phase. These problems are primarily due to the open-loop structure often being followed in software development. It is straightforward that relying on human intervention to tune a system is not feasible due to the conditions changing constantly, rapidly and unpredictably. Therefore, there is a high demand for management complexity reduction, management automation, robustness, and achieving all of the desired quality requirements within a reasonable cost and time range during operation. We believe that self-adaptive software systems is a response to these demands; it is a closed-loop system with a feedback loop aiming to adjust itself to changes during its operation. This allows them to automatically find the best way to accomplish a given goal with the resources at hand. This capability would benefit the full range of computing systems, from small devices to grids of computing systems.

This dissertation first aims at studying about the architecture of a self-adaptive software system. Second, in this scope of this dissertation a framework called Application Heartbeats will be studied as a monitoring infrastructure for building self-adaptive software. This framework helps in providing methods for an application to define its goals,to observe its current states and performance. We then going to provide to readers the imple-

mentation logic of common components in a self-adaptive software system. Taking those knowledges as foundation, we then discuss into details about implementing a self-adaptive software system. During dicussion, we are going to propose the concept of running configuration as a unification for using implementation library and parameter API library. The concept of running configuration have the very similar meaning to the concept of context used in pervasive system. It helps programmers in expressing the action self-adaptive software system should take corressponding to each situation. Finally, we will illustrate the dicussed methodology by the implementation of self-adaptive software application. To demonstrate the fact that self-adaptive property has a wide spectrum of benefits, we chose Android mobile operating system as the development environment. Our project aims at build a small self-adaptive software application - an MP3 decoder running on Android. As a self-adaptive software application, this application has to have the ability to monitor its own progress or delegates this to external observer and to adapt itself to the new situation. During implementation, all dicussed methods will be experimented. The application finally is put to test to verify the benefit of self-adaptive propery.

The remainder of this dissertation is structured as follows. In Chapter 1, a brief introduction over self- awareness adaptive software systems in relation with abroader concept - `autonomic system` is provided to reader. A detailed study about self-adaptive software system is introduced in Chapter 2. Chapter 3 is intended to provide to reader about Application Heartbeats framework, and the methodology to build an self-adaptive software

system. Chapter 4 introduces to readers the Android mobile operationg system as a developing environment and design in detail of our project to illustrate the methodology in developing a self-adaptive software system. Chapter 5 represents to readers a set of results to demonstrate the benefits of proposed approach. Finally, Chapter 6 concludes the document by a proposal of some possible future works.

Contents

List of Figures

List of Tables

List of Listings

List of Abbreviations

2D	Two Dimensions
3D	Tree Dimensions
API	Application programming interface
BSD	Berkeley Software Distribution
CPU	Central Processing Unit
DARPA	Defense Advanced Research Projects Agency
DVM	Dalvik Virtual Machine
FPU	Floating Point Unit
ID	Identifier
JNI	Java Native Interface
JVM	Java Virtual Machine
MAPE-K	Monitor-Analize-Plan-Execute loop

MP3	Moving Picture Expert Group-1/2 Audio Layer 3
ODA	Observe-Decide-Act loop
PCM	Pulse-code Modulation
PID	Process Identifier
QoS	Quality of Service
SDK	Software Development Kit
SMS	Short Message System

Chapter 1

Introduction

This Chapter is going to introduce to readers one of the most common problem that plagues moderning computing systems, the skyrocketing complexity problem. On the way studying the solutions for this problem, scientists and engineers have made significant efforts to design and to develop systems which can change their behaviors without human intervention and adapt itself to the unpredicted changes in environments or also being called in short self-adaptive system. While self-adaptive systems are used in a number of different areas, in this dissertation we only focus on their applications in the software domain, called self-adaptive software. This Chapter will provide an overall introduction on this topic including basic concepts which will be refered throughout all other chapters.

1.1 Introduction to the problem

Computing systems now can be seen as an aggregate of complex computer architectures and software systems. According to Moore's law[1]:

> *"The number of transistors that can be placed inexpensively on an integrated circuit has doubled approximately every two years[1]"*

This implies that we have skyrocketing in the evolution of computer architecture's complexity. Moreover, this steady evolutionary trend in hardware creates more computational power availability and fosters the development of software representing in the creation of many complex software systems. Designing and developing application in modern world is now not a trivial process. Besides, the complexity of software systems makes the (re-)configuring, troubleshooting, and in general maintaining tasks costly and time-consuming during the operating phase. Therefore, there is a high demand for management complexity reduction, management automation, robustness, and achieving all of the desired quality requirements within a reasonable cost and time range during operation[2]. We believe that self-adaptive computing systems, and thus self-adaptive software systems is a right answer to these demands.

1.2 Research context

This section aims at answering the question : why we need self-adaptive software. One of the most important reason which has been introduced in

previous section, is the increasing cost of handling the software systems' skyrocketing complexity to achieve their desired goals[3]. Different software systems have different definitions of their goals, some deal with management complexity, robustness in handling unexpected condition (e.g., fault tolerance) changing priorities and policies governing the goals, and changing conditions (e.g., in the context of mobility)[2]. Traditionally, software development focuses on handling complexity and its internal quality attributes. However, recently due to an increase in the heterogeneity level of software components, frequent changes in context/goals/ requirements during runtime, and higher security needs, there has been an increase demand for software systems to be able to deal with issues those happen at operation time. In fact, some of these causes are consequences of the higher demands for ubiquitous, pervasive, embedded, and mobile applications, mostly in the Internet and ad-hoc networks[2].

Self-adaptive software is expected to fulfill those requirements at runtime in response to changes. However, managing software at runtime is often costly and time-consuming due to the fact that it is impossible to make any assumption about the contexts where software is going to operate in nor the mutation between those contexts in timing manner. Therefore, an adaptation mechanism is expected to be smart enough to detect changes and take appropriate actions accordingly at a reasonable cost and in a timely manner. It is dynamic/runtime change which is the basis for adaptation in self-adaptive software systems.

As being mentioned above, scientists and engineers in Software En-

gineering area have made significant efforts to design and develop self-adaptive software systems. These systems address adaptivity in various concerns including performance, security, and fault tolerance [4]. Several methods have been proposed to incorporate adaptation mechanism into software systems. It is straightforward that softwares which are normally being implemented as an open-loop system, should be converted to a closed-loop system using feedback mechanisms. While adaptivity may be achieved through feed-forward mechanisms as well (e.g., through workload monitoring), the feedback loop takes into account a more holistic view of what happens inside the application and its environment[2]. Figure 1.1 shows the adaptation loop or often called ODA loop (ODA stands for Observe-Decide-Act) of a general automic system.

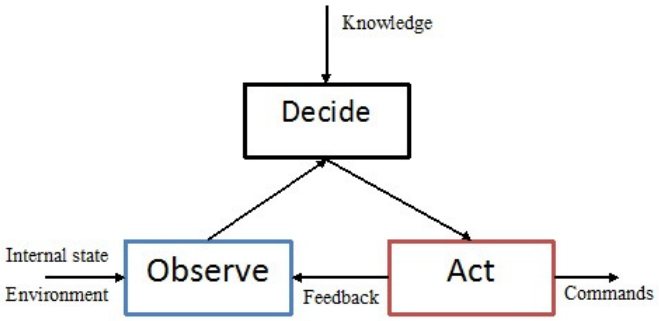

Figure 1.1: Observe-Decide-Act loop

As can be observed, there are 3 fundamental components in an ODA loop :

- `Observe component`: takes part in monitoring both the internal issues and context environment using sensing sub-system.
- `Decide component`: bases on observed information and the system's goal definitions, this component uses logic system to make decisions of which action shoud be taken next.
- `Act component`: translates the decision made by the Decide component into a set of actions which are going to be executed by actuators.

A self-adaptive software means that the whole body of the software will be implemented in serveral layers while the context encompasses everything in the operating environment that affects the system's properties and its behaviors. Thus, self-adaptive software is a closed-loop system with feedback from both the internal and the environment.

It is obviously that self-adaptive software systems have strong relationship with autonomic and self-managing systems [5], thus it is difficult to draw a distinction between these terminologies. Many researchers use the terms self-adaptive (not specifically self-adaptive software), autonomic computing, and self-managing interchangeably[2]. However, from one point of view, the self-adaptive software domain is more limited, while autonomic computing has emerged in a broader context. This means self-adaptive software has less coverage and falls under the umbrella of autonomic computing. From another point of view, we can consider a layered model for a software-intensive system that consists of: application(s), middleware, network, operating system, hardware [6], and sublayers of middleware [7]. According to this view, self-adaptive software primarily covers

the application and the middleware layers, while its coverage fades in the layers below middleware. On the other hand, autonomic computing covers lower layers too, down to even the network and operating system . However, the concepts of these domains are strongly related and in many cases can be used interchangeably[2].

1.3 Hierarchy of adaptation properties

This section answers the question about which properties that an autonomic system in general and an self-adaptive software system in particular should contain. It is IBM, through their research [8] specified the initial well-known set of adaptation properties or often called self-* properties. There are eight properties which are depicted in the hierarchical structure in Figure 1.2 [1].

Those eight properties are categorized in three levels. The most general level contains self-adaptiveness and self-organizing properties, and they will be decomposed into major properties and primitive properties in the two corresponding lower level:

- `General Level`: This level contains global properties of self-adaptive software. A subset of these properties, which falls under the umbrella of self-adaptiveness [10], consists of self-managing, self-governing, self-maintenance [5], self-control [11], and self-evaluating [12]. Another subset at this level is self- organizing [13], [14], which emphasizes decentralization and emergent functionality(ies). The self-organizing

[1] Graphic taken from the thesis of Marco Triverio[9]

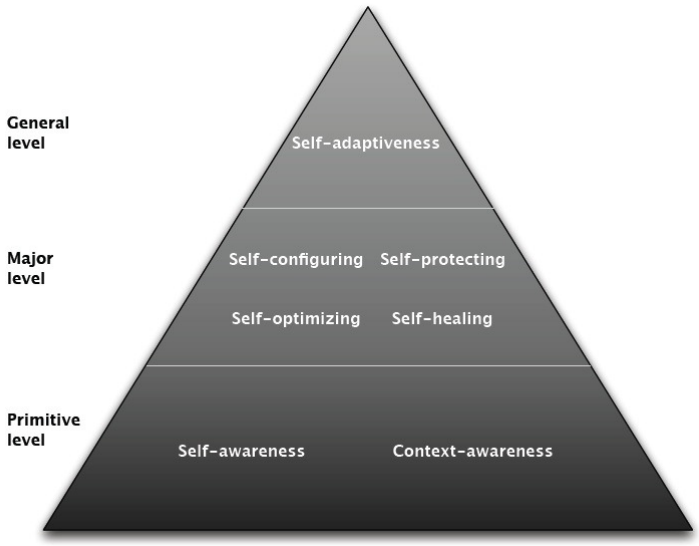

Figure 1.2: Adaptation Hierarchy

property is bottom-up, in contrast to self-adaptiveness, which is typically top-down.

- `Major Level`: This level contains four main property which have been defined in accordance to biological self-adaptation mechanism[5]. The details of each property is depicted below:
  - `Self-configuring`: This property means the capability of system to configure itself automatically and dynamically. This not only means the system can (re)configure its existing components to adapt new situation, but also mean the capability of installing, updating, and composing/decomposing new software components.
  - `Self-healing`: This property express the capability of system to self-discover, to self-diagnose, and to react against failures. In

a smart system, this also means the ability of system to foresee the potential problem to take appropriate action to prevent disruptions. Self-diagnosing provide the ability to diagnose errors while self-repairing refers to the ability of recovering from them.

  - `Self-optimizing`: It is straightforward that the algorithm on which software systems implemented are not closed but provides a set of parameters which can be tuned to get a better performance in different operating contexts. Self-optimizing express the ability of self-managing resource by adjusting parameters in order to get the desired goals. Response time, throughput, utilization, and workload are examples of important requirement related to this properties.

  - `Self-protecting`: is the ability of automatically defending against malicious attacks or failures which maybe using early monitoring to anticipate threats or mitigate their affects.

- `Primitive Level`: Self-awareness, self-monitoring, self-situated, and context- awareness are the underlying primitive properties. Self-awareness means that the system is aware of its self states and behaviours based on self-monitoring. In contrast, Context-Awareness means that the system is aware of its context in which it is operating in. These reflect our vision in previous section indicated that software system contains a closed-loop with feedback both from the internal and the environment.

1.4 Summarization

In this chapter, we have already introduced to readers the skyrocketing complexity problem of computing systems. Scientists and engineers on the way studying the solution for this problem have proposed self-adaptive computing system as the full of promise answer. We pointed out that self-adaptive system can help in improving the performance, customizability, extensibility, maintainability, availability and security while reducing the effort and skills needed to take control and manage the system. We also discussed about the fundamental properties of a self-adaptive computing system and categorized them using hierarchical structure. It is important to notice that self-adaptive computing has emerged a broader context than self-adaptive software but it is sometime difficult to draw the distinction between them. In the following chapters, we are going to discuss more about self-adaptive software systems rather than the general self-adaptive computing systems but most of described concepts can be used interchangeably without losing their validity.

Chapter 2

Self-Adaptive Software Systems

This chapter is intended to provide to readers a more detailed view about self-adaptive software systems. Firstly, we are going to introduce some definition of self-adaptive software system. Secondly, we will discuss about what the important requirements of a self-adaptive software system are, and how to elicit those requirements. Thirdly, a completed and unified taxonomy of self-adaptive software systems will be represented to introduce to readers the general view of self-adaptive software's world. Finally, we are going to discuss about control loop models which are the fundamental elements of the self-adaptive software systems.

2.1 Definitions

There are alot of definitions of self-adaptive software which are proposed by many researcher so far. Among them, one is provided in a DARPA Broad Agency Announcement [15] indicates that :

> *"Self-adaptive software evaluates its own behavior and changes behavior when the evaluation indicates that it is not accomplishing what the software is intended to do, or when better functionality or performance is possible.[15]"*

A similar definition also was given by Oreizy in his work [10]:

> *"Self-adaptive software modifies its own behavior in response to changes in its operating environment. By operating environment, we mean anything observable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation.[10]"*

They can be considered as some of formal concepts of self-adaptive software. Prior to those, there has been a related point of view which based on the apdative programming language principle. Adaptive programming language is an extension of objecte-oriented programming language which is defined in [16]:

> *"A program should be designed so that the representation of an object can be changed within certain constraints without affecting the program at all.[16]"*

Softwares which are implemented using multi switchable logic layers can be considered belong to adaptive programming. Although, up to now many definitions have been proposed, all self-adaptive systems have the common point that is their life cycle wil be continued after installation,

gather and evaluate context information and change their behaviors to adapt the new context at all time.

2.2 Adaptation Requirements Elicitation

In Software Engineering it is obviously that to start building a software, requirement analysis is the first and the most critical step which guarantees the success of the project. For a self-adaptive software systems, beside of those analyses which we have to carefully accomplish for a general software, it is important that we have to address the adaptation requirements. Many researches have proposed methods that can be used in elicitation the essential requirements of self-adaptive software. Among of them, [3] proposed six important questions to partially address these requirements and they were modified and completed in [2]. The list below shows the question set depicted in [2]:

- `Where`: This set of questions need to be answered to specify where the need for change is. Which artifacts at which layer (e.g., middleware) and level of granularity need to be changed? Thus, it is required to collect information about attributes of adaptable software, dependency between its components and layers, and probably information about its operational profile. Therefore, The answers to `Where` questions set help in locating the problem that needs to be resolved by adaptation.

- `When`: This set of questions related to the temporal aspect, mostly

specifies when a change need to be applied, and when it is feasible to do so(e.g., it's dangerous to swap components using hot swap method without reaching the quiescent state). It is straightforward that there exist some constraints on applying changes on a running software system to maintain its consistency. This set of questions also requires answers related to the frequency of change like how often does the system need to be changed? or are the changes happening continously? .etc.

- `What`: The answer to this set of questions identifies what attributes or artifacts of the system can be changed through adaptation actions, and what needs to bechanged in each situation. These can vary from parameters and methods to components, architecture style, and system resources. It is also important to identify the alternatives available for the actions and the range of change for attributes (e.g., parameters). Moreover, it is essential to determine what events and attributes have to be monitored to follow up on the changes, and what resources are essential for adaptation actions.

  The distinction between the `what` and `where` questions is notable. `Where` addresses which part of the system causes the change, while `what` refers to the attributes and artifacts that need to be changed to resolve the problem. Sometimes, the entity that is the source of change is also the entity that needs to be changed (e.g., component swapping). Therefore, although these questions are related, they address different aspects of adaptation.

- `Why`: This set of questions deals with the motivations of building a self-adaptive software application. These questions are concerned with the objectives addressed by the system (e.g., robustness). If a goal-oriented requirements engineering approach is adopted to elicit the requirements, this set of questions identifies the goals of the self-adaptive software system.

- `Who`: This set of questions alter the level of automation thus the level of human involvement in self-adaptive software.

- `How`: One of the important requirements for adaptation is to determine how the adaptable artifacts can be changed and which adaptation action(s) can be appropriate to be applied in a given condition. This includes how the order of changes, their costs and after-effects are taken into account for deciding the next action/plan(e.g., hotswap method is a very promising answer to this set of question in self-adaptive software area).

Not like general softwares which only need requirement analysis at developing phase, self-adaptive softwares need requirement analysis at both developing phase and operating phase. This comes from the fact that self-adaptive softwares contain a closed-loop with feedback from context. In developing phase, analyses will be undertaken for building self-adaptive software either from scratch or by re-engineering a legacy system. During this phase, designers elicit the requirements based on above question in order `to build adaptable software` (e.g. software will be designed to contain serveral layers) and setup mechanisms to be used at the operat-

ing phase. At the operating phase, the software is required to manange the feedback information `to adapt itself to changes` in the self/context of software application based on these questions. The questions in this phase are mostly being asked about where need a change, what need to be changed, and when and how it is better to be changed.

2.3 A taxonomy of self-adaptation

This section is going to provide to readers a taxonomy of self-adaptation. There are many way for categorizing based on different aspects have been proposed so far. Some of them start from static approaches and then moves on to dynamic ones [10], while others tend to focus on techniques and technologies [6]. In their work [2], Mazeiar Salehei and Ladan Tahvildari from University of Waterloo uninified these classifications into a taxonomy depcited below, and also introduced new facets to fill the gaps:

According to [2], their proposed classification was taken in the relationship with the method of eliciting adaptation requirements using six important questions. In fact, it is impossible to map one by one between each questions set and one facet of classification. As depcited in Figure 2.1, the first level of hierachical taxonomy contains: Object to adapt, realization issues, temporal characteristics, and action concerns.

- `Object To Adapt`: This facet is in the relationship with `What` and `Where` questions set. It is going to be devided in to three smaller subfacets indicating the main object for adaptation:

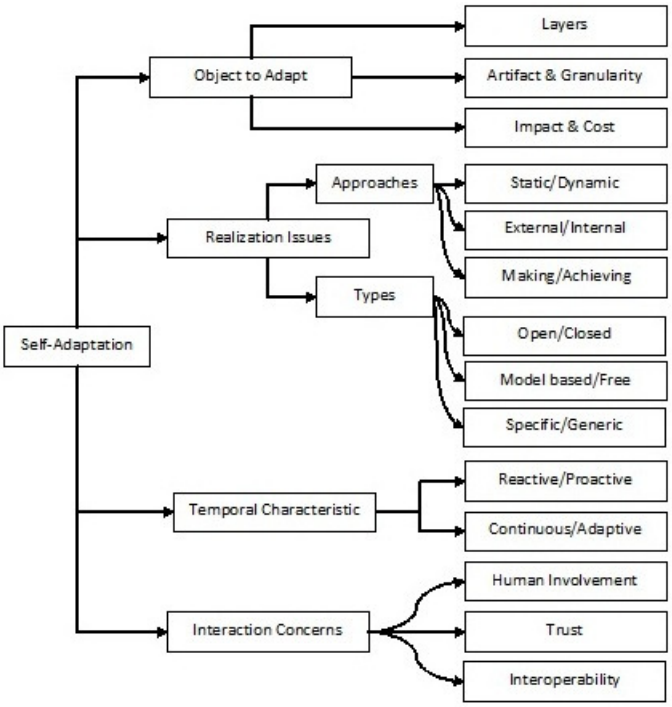

Figure 2.1: A self-adaptation taxonomy

  - `Layer`: This sub-facet specify the layer which causes problem or the layer which can be changed. Adaptation action can be applied on different layers.
  - `Artifact and Granularity`: This sub-facet indicate software components and the levels of granularity can be changed by adaptation actions. The adaptation actions can take place either on modules or on the architecture and the way they are composed. An application can be decomposed into components depending on the architecture and granularity level and all of them are subject of changes, thus object of adaptation process.
  - `Impact and Cost`: The Impact aspect describes the scope of effect after adaptation actions are operated, while Cost aspect refer to the price for taking those adaptation actions(e.g., execution time, resources, algorithm complexity). They can be further divided in to week and strong classes. Weak adaptation more or less means the action of tuning parameters which can be performed at low-cost with limited-impact. Besides, Strong adaptation deals with the alters software artifacts which are performed at high-cost with great impact. It is notable to notice that a Strong class is a composition which may contains other Strong or Weak classes.

- `Realization Issues`: This facet are deal with the questions set `How`, and being devided into two smaller class called `Approach` and `Type`. This classification means the issue of how the adaptation can

be applied is realized either by their approaches or their types.

- `Approach`: This sub-facet of taxonomy indicate the approach of incorporating adaptivity into the system. The following sub-facets can be identified:
  - Static/Dynamic Decision Making: This subfacet classifies two kind of decision making support mechanism : static and dynamic. In the static option, the decision making mechanism is hard-coded (e.g., as a decision tree) and its modification requires recompiling and redeploying the system or some of its components. In dynamic decision-making, policies [17], rules [18] or QoS definitions [19] are externally defined and managed, so that they can be changed during runtime to create a new behavior for both functional and non-functional software requirements.
  - External/Internal Adaptation: This appproach takes classification based on the level of separation between adaptation mechanism and application logic. Internal means the strong couple between application and adaptation logic mostly based on programming language features(e.g., conditional expressions, functional programming). It is obviously that using this approach leads to the poor scalability and maintainability and only suitable for simple software systems to handle local adaptation. Recently, some programming languages have been extended to contain the notion of `Context` and intro-

duce the definition of `Context-Oriented Programming`, to provide the ability to access global information about the system and correlating events happening in its self/context.

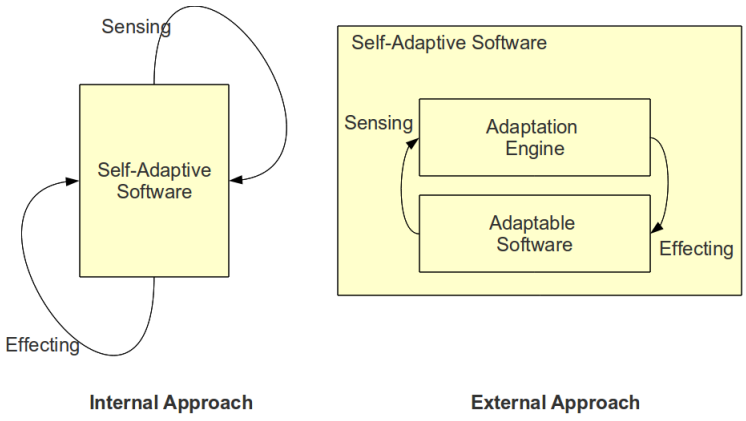

Figure 2.2: The internal/external approaches

External approaches use an external adaptation engine containing adaptation processes. Drawing the distinction line devides the self-adaptive software system into an adaptation engine and an adaptable software. The decoupling between application logic and adaptation logic fixed the drawbacks of internal approaches. Moreover,a significant advantage of the external approach is the reusability of the adaptation engine, or some realized processes for various applications. This means that an adaptation engine can be customized and configured for different systems. The Figure 2.2 describes those approaches for building self-adaptive software system.

* Making/Achieving Adaptation: This subfacet indicate two strategies which can be used to introduce self-adaptivity into software system. The first strategy is to engineer self-adaptivity into the system at the developing phase. The second strategy is to achieve self-adaptivity through adaptive learning. They call these two approaches making and achieving in [20]

– `Type` : Another important facet is the type of adaptation. It specifies whether the adaptation is open or closed to new alternatives, whether it is domain specific or generic, and whether it is model-based or model-free.

* Close/Open Adaptation: A close-adaptive system has only a fixed number of adaptive actions, and no new behaviors and components can be introduced during runtime. Whereas, in open adaptation, self-adaptive software can be extended, and consequently, new components can be added, and even new adaptable entities can be introduced to the adaptation mechanism

* Model-Based/Free Adaptation: In model-free adaptation, the mechanism does not have a predefined model for the environment and the system itself. In fact, by knowing the requirements, goals, and alternatives, the adaptation mechanism adjusts the system. On the other hand, in model-based adaptation, the mechanism utilizes a model of the system and its context. This can be realized using different model-

ing approaches, such as a queueing model for self-optimizing, architectural models for self-healing [21], or domain-specific models in [22].

    * Specific/Generic Adaptation: This subfacet divides two type of adaptation based on their applied domain. Specific adaptation indicates solutions which address for a specified set of domains, whereas generic adaptation is valid solutions for a generic domain which can be configured using policy setting, alternative components or set of adaptation processes for different domains.

- `Temporal Characteristics`: This facet deals with answer for `When` questions set. The following subfacets can be identified:
  - `Reactive/Proactive`: In the reactive mode, the system responds when a change has already happened, while in the proactive mode, the system predicts when the change is going to occur.
  - `Continuous/Adaptive Monitoring`: This subfacet captures whether the monitoring process is continually collecting and processing data or being adaptive in the sense that it monitors a few selected features, and in the case of finding an anomaly, aims at collecting more data.
- `Action Concerns`: This facet make classification based on the level of interaction between one self-adaptive system and other self-adaptive

system or human through its interfaces.

- `Human Involvement`: In self-adaptive software, human involvement can be discussed from two perspectives. First, the level of automation, and second, how well it interacts with its users and administrators. There are notable difference between two perspectives. The former indicates that human involvement is not desirable, whereas the latter indicates the quality of human interaction. According to the latter, human involvement is essential and quite valuable for improving the manageability and trustworthiness of self-adaptive software.
- `Trust`:Trust is a relationship of reliance, based on past experience or transparency of behavior. Trust can be either watched from two different perspectives: security issues and the extend to which a system can be believed to correctly adapt.
- `Interoperability Support`: Self-adaptive software often consists of elements, modules, and subsystems. In open approaches interoperability is always a concern because system's elements need to be coordinated with each other to have the desired self-* properties.Especially in the area of distributed complex systems, local adaptation and interoperability support mechanisms are needed to provide global adaptation which is a critical requirement.

2.4 Control Loop Models

It is obviously that self-adaptive aspects of software-intensive systems can be hidden within the system design, but what self-adaptive software systems have in common is that:

- Typically design decisions are partially made at runtime.
- The systems reason about their state and environment using feedback mechanism.

The feedback mechanism contains a closed loop plays an important role in self-adaptive software architecture. This often is called control loop or adaptation loop because of enabling adaptivity to the systems. In Chapter 1, we mentioned about the ODA control loop which is represents a feedback mechanism with three processes. Another similar control loop was proposed in [5] and called the MAPE-K loop. This loop consists of four processes, as well as sensors and effectors and being used in the context of autonomic computing. Figure 2.3 depicts four adaptation processes including the Monitoring, Analyzing, Planning and Executing functions, with the addition of a shared Knowledge-base. In fact, the adaptation loop proposed by [5], which exist at the operating phase of an adaptive-software system is similar to the ODA control loop for general autonomic system described in Chapter 1 and can be summarized as follows:

- `The monitoring process` is responsible for collecting and correlating data from sensors and converting them to behavioral patterns

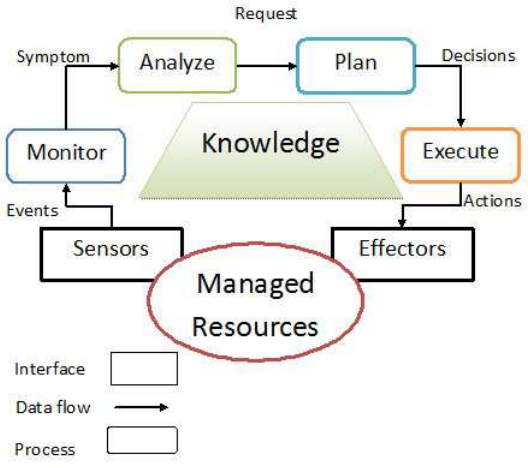

Figure 2.3: The MAPE-K loop

and symptoms. The process can be realized through event correlation, or simply threshold checking, as well as other methods.

- `The analyze process` is responsible for analyzing the symptoms provided by the monitoring process and the history of the system, in order to detect when a change (response) is required. It also helps to identify where the source of a transition to a new state (deviation from desired states or goals) is.

- `The plan process` determines what needs to be changed, and how to change it to achieve the best outcome. This relies on certain criteria to com- pare different ways of applying the change, for instance by different courses of action.

- `The execute process` is responsible for applying the actions determined by the deciding process. This includes managing non prim-

itive actions through pre- defined workflows, or mapping actions to what is provided by effectors.

Sensors and effectors are essential parts of MAPE-K loop. Sensors play an important role in collecting software's internal state data and context information, while effectors work the role like `Act` in ODA control loop in applying changes. In fact, the first step in realizing self-adaptive software is instrumenting sensors and effectors. One notable point of missing in MAPE-K loop is that the input for being monitored by sensors in this control loop is not specified. In fact, system's internal state and context information are implied to be the implicit sources for sensors.

In [23], a similar control loop is also presented but directed in the context of autonomic communication. This adaptation loop consists four process: monitoring, detecting, deciding and act. Figure 2.4 depicts an overview of the main activities around the control loop but ignores the properties of the control and data flows around the loop.

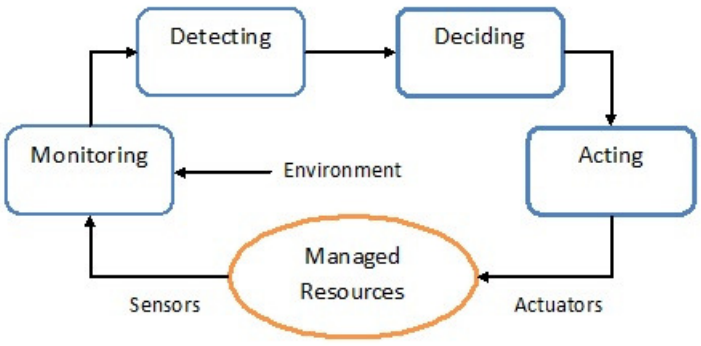

Figure 2.4: The Dobson adaptation loop

The feedback cycle starts with the collection of relevant data from environmental sensors and other sources that reflect the current state of the system. Next, the system analyzes the collected data. There are many approaches to construct and to reason about the raw data (e.g., using applicable models, theories, and rules). The system then makes a decision about how to adapt in order to reach a desirable state. Approaches such as risk analysis can help make this decision. Finally, to implement the decision, the system must act via available actuators and effectors. A formal requirments elicitation method using questions set discussed above can be useful in analyzing the requirment for implementing each adaptation process of the loop.

2.5 Summarization

This chapter has provided to readers fundamental concepts in self-adaptive software area. We first presented some formal definitions, introduced methods for eliciting requirments, dicussed about the methods to classify self-adaptive software systems based on various aspects. We then discussed about the most important part of this kind of software system - the control loop. In the next chapter, we are going to represent more details about how to experimentally create a self-adaptive software and verify its benefits.

Chapter 3

Methodology

We have dicussed about self-adaptive software in previous chapter. This chapter aims at providing a experimental view of the self-adaptive software's implementation. We will first take a dicussion about a framework called Application Heatbeats which help in defining application's desirable goals and monitoring system's states. This chapter next explores general methods for implementing self-adaptive software components. At the end of this chapter, we are going to discuss about the Consensus Object implementation as the center element which connects other components to create the complete architecture of self-adaptive software systems.

3.1 Application Heartbeats Framework Introduction

In previous chapter, some type of control loops have been explored. One notable common point between them can be recognized that they all need a mechanism of instrumentation. Many solutions have been proposed so far

providing good flexibility in monitoring system's internal state and context information but with different expenses of simplicity and portability. Starting from the idea of providing a simple, portable and general method of monitoring an application's actual process towards its goals and making that information available to external observers, Application Heartbeat Framework has been proposed in [24] implementing a very simple yet extremely powerful monitoring infrastructure. A key factor has been introduced in this framework is application's heartbeat which is very similar to human heartbeat. By allowing applications to inform their heartbeat and using monitoring infrastructure to monitor them, Application Heartbeats framework helps programmers to express their applicationâs goals and monitor the application's progress using a simple set of APIs.

As being mentioned, Application Heartbeat framework aims at the simplicity and portability, that is why it is written in C using basic system calls which are supported by most of compilers and can be callable from both C and C++. Table 3.1 lists all support functions by the framework.

Technically, when the HB_heartbeat function is called, a new entry containing a timestamp, tag and thread's ID is written into a file. One file is used to store global heartbeats. When $per - thread$ heartbeats are used, each thread writes to its own individual file. It is mandatory to define a private directory for framework to store those file and it is done by using environment variable called HEARTBEAT_ENABLED_DIR. A mutex is used to guarantee mutual exclusion and ordering when multiple threads attempt to register a global heartbeat at the same time. When an external

Table 3.1: Heartbeats API functions

Function Name	Description
heartbeat_init	Initialize the Heartbeat runtime system and specify how many heartbeats will be used to calculate the default average heart rate.
heartbeat_finish	Do the cleanup stuff
hb_get_current	Returns the average heart rate calculated from the last window heartbeats.
hb_get_history	Called to retrieve the n recently heartbeats history
hb_get_global_rate	Called to retrieve the global target heart rate
hb_get_windowed_rate	Called to retrieve the windowed target heart rate
hb_get_min_rate	Called to retrieve the minimum target heart rate
hb_get_max_rate	Called to retrieve the maximum target heart rate
heartbeat	Generate a heartbeat to indicate progress

application wants to get information on a $heartbeat - enabled$ program, the corresponding file is read. The target heart rates are also written into the appropriate file so that the external applications can access them. The snip code 3.1 depicts the structure used to store heartbeat's information:

```
1  typedef struct {
2    int64_t first_timestamp;
3    int64_t last_timestamp;
4
5    int64_t* window;
6    int64_t window_size;
7    int64_t current_index;
8
9    int steady_state;
10   double last_average_time;
11
12   heartbeat_record_t* log;
13
14   FILE* binary_file;
15   FILE* text_file;
16   char filename[256];
17
18   pthread_mutex_t mutex;
19
20   HB_global_state_t* state;
21 } heartbeat_t;
```

List 3.1: Sample of a Heartbeat's structure

This framework also provides an extension set of APIs called Heart Rate Monitor which supports external applications in monitoring Heartbeats-enabled applications to get target heart rates. However,this implementation does not support changing the target heart rates from an external ap-

plications. The Table 3.2 depicts the APIs which can be called from external application (e.g., adaptation engine):

Table 3.2: Heart Rate Monitor API functions

FunctionName	Description
heart_rate_monitor_init	Preparation step to get access to specific process heartbeats information.
heart_rate_monitor_finish	At this moment this function do nothing.
hrm_get_current	Allows external application to get the current heart rate of specific process.
hrm_get_history	Allows external application to retrieve n recently heartbeats history
hrm_get_global_rate	Allow external application to retrieve the global heart rate of specific process
hrm_get_windowed_rate	Allow external application to retrieve the windowed heart rate of specific process
hrm_get_min_rate	Allow external application to retrieve the minimum heart rate of specific process
hrm_get_max_rate	Allow external application to retrieve the maximum heart rate of specific process
hrm_get_window_size	Allow external application to retrieve the default window size of specific process

A general execution scenario using Application Heartbeat framework will be summarized as below :

- $Hearbeats - enabled$ Applications: calls to HB_heartbeat function are injected into application code at significant points to express the application's progress. Normally, a hearbeat log record contains current time and thread's ID of caller. It may also contain a tag which is inserted by the users to provide additional information. A call to HB_init should be invoked prior to all calls to HB_hearbeat to setup stuff. Default window size may be the most important setup information needs to be specified during initialization step because it alters the number of heartbeats will be used to calculate the moving average rate. The HB_get_current function will return the average heart rates for the most recent heartbeats; by saying recent we mean the heartbeats belong to the window. When more $in - depth$ analysis of heartbeats are required, the HB_get_history function can be used to get a complete log of recent heartbeats. It returns an array of the last n heartbeats in the order that they were produced.

- External application: An external application may monitor $Heartbeats - enabled$ using Heart Rate Monitor APIs. It is mandatory for external application to know the identification of $Heartbeats - enabled$ thread it wants to monitor. A call to heart_rate_monitor_init function with target's ID will provide a handle to access target's heartbeat information. Succesfully in initializing step will let external application extract heartbeat information on the moving.

The Figure 3.1 [1] depicts two scenario of using Application Heartbeats

[1] Graphic taken from the thesis of Marco Triverio[9]

framework : (1) $self - optimizing$ application and (2) Optimization by an external observer.

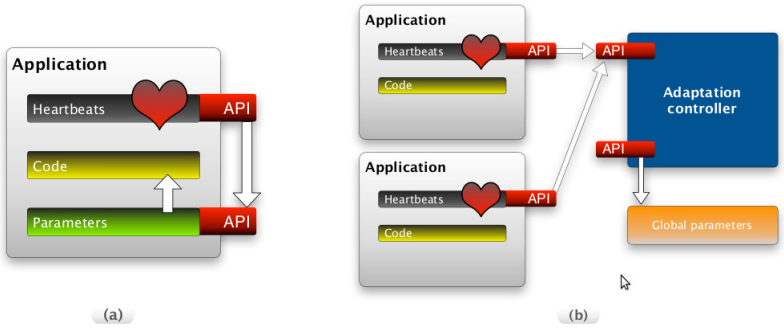

Figure 3.1: Two scenario of using Application Heartbeats framework

Having a simple, standardized API makes it easy for programmers to add Heartbeats to their applications. It also crucial for portability and interoperability between different applications, runtime systems, and operating systems. Registering goals with external systems enables optimizations that are unavailable within an application such as modifying scheduling decisions or adjusting hardware parameters. When running multiple Heartbeat-enabled applications, it also allows system resources (such as cores, memory, and I/O bandwidth) to be reallocated to provide the best global outcome [24].

3.2 Control loop implementation

In Chapter 2, we have already dicussed about some control loops. Among them, ODA loop is the most general one and theoretically a self-adaptive software system can be implemented using such loop model.

3.2.1 Logic of Observe process

As being mentioned in Chapter 2, Observe process makes a system aware of its internal state and context around. In [9], three levels of awareness have been introduced:

- Awareness through the data gathered by sensors: For the first kind of awareness, a monitoring mechanism is needed and a study about the Application Heartbeats framework in previous section makes us believe that it is the right choice for this purpose. Application Heartbeats framework is simple but a powerful framework that allows software components (such as applications) to assert performance goals and keep track of the progress.

- Awareness of the availability and potential of actuators: The second kind of awareness requires system to have a full knowledge about all available actuators and their level of potential. By saying level of potential, we mean that the system need a infrastructure to evaluate actuators' impact and cost. The impact aspect describes the scope of effect by specific actuator, while cost aspect refers to the price for using that actuator.

- Awareness awareness of the possible targets of actions: This kind of awareness requires the possible targest aimed by each specific actuator must be defined. In static sense, possible target can be software implementation layers, while in dynamic sense, information about possible target of actions might change dynamically one or more ser-

vices oriented mechanism can be introduced as the target manager.

3.2.2 Logic of Decide process

The decision engine stands at the center of the ODA loop. It is the element that exploits the awareness given by the observation mechanisms to elaborate a plan for future behavior. The decision process can be implemented in reality either in static way or dynamic way. In static option, decision making mechanism is hard-coded and its modification requires recompiling and redeploying the system or some of its components. Whereas, in dynamic option, policies, rules or Qos definition are externally defined and managed so that they can be changed during runtime. Methods for implementing Decide process mainly come from the general area of Datamining and Knowledge Extraction. It's hard to indicate which method is the best, the chosen method is decided in the corelation with system's requirments and can be summarized as below:

- Decision trees: They are effictiently evaluated at runtime but it is hard to extend them because they are often static, and being $hard - coded$.
- Rule-based: This method allows for a good degree of flexibility if the operating environment is well known and can be described through rules. Rules guarantee a fairly deterministic behavior and allow actions to be taken in response to the occurrence of an event.
- Pattern-matching : It is highly static and used to make the program attune to a context that is ever-changing with some periodic behavior.

- Reinforcement learning: is a $sub-area$ of machine learning. Decisions made previously will be evaluated and provide knowledge for taking current decision.

- Game theory: this kind of AI also can be used to implement a $non-centerlized$ system

3.2.3 Logic of Act process

The Act process is implemented to translate the selected decision in to a set of actions which can be performed by available actuators. The implementation of this process does not contain the implementation of actuators. In fact, acting process's logic aims at managing available actuators, collecting knowledge about the ability each actuators provide through their exposed interface, translating selected decision into a set of primative/non-primative actions through pre-defined workflows or mapping actions to what is provided by actuators.

3.3 Services

In [9], instead of the concept of actuators, an extension concept is proposed. By saying an extension we mean that new concept not only represents a component capable of performing changes on one or more applications or on the system but it also includes the interact to interact with other components(e.g, the Consensus Object, application,etc.). There is a certain amount of elements that can affect performance on a system. They come

in different forms and they bring different consequences. [9] has made a summarization of services base on consequences:

- Application knobs: They are parameters in an application that can be changed, causing an alteration in performance.
- Application implementations: Many applications make use of a number of algorithms to carry out their duties. They are considered as application's layers and depending on the scenario a different implementation might lead to better results. For this reason changing the algorithm to adapt the change in scenario may alter the performance.
- Core allocation: Under modern operating system it is possible to link a given process to a certain processor or core. By doing this we can take full adavantages of cores, there will be no need for time sharing among processes.
- Memory allocation: The core allocation service is particularly useful when used for processes that are $CPU - intensive$. There exist another category of processes that requires, instead of processing resource, memory. Such programs are said $memory - intensive$: in order to be optimized they require memory to be allocated easily and efficiently. This optimization is enabled on a $per - application$ basis.
- Niceness adjusting: Modifying the niceness (which is the priority of a process) can enhance or reduce performance of a given application. This optimization is enabled on a $per - application$ basis.

- Frequency scaling: It is a widely popular service that reduces processor clock frequency for slowing down the computation on the entire system.

Because application's knobs are part of application, any change of them will be vision to application. Whereas, the change in other kinds of service will not or should not be visible to application. Utilizing services belong to the second category enable a one-to-many and many-to-many relationship between application and services.

3.4 Consensus Object

3.4.1 Introduction

We have already mentioned about those logics should be implemented by each process. In experimental implementation, this means we need a component that does the following tasks:

- Discovers and dynamically updates availability and characteristics of actuators.
- Discovers and dynamically updates the possible targets aimed at by the actuators.
- Analyzes data coming from the applications, decides which applications need to be targeted, chooses which actuators to activate or to modify, and submits the plans to the actuators.

In [9], such component is described and called the Consensus Object. It is clear that because the Consensus Object needs to have the global view of the system, it has to be implemented as a center elemenent in the control loop. This leeds to the fact that the Consensus Object becomes a single point of failure. To prevent the failure of the Consensus Object cause effect to the whole system, actuators or being called in [9] as services should be implemented independently from the Consensus Object. Moreover, as a center element, it is required to expose a standardized interface so that it can easily be intracted by different actuators. In fact, Consensus Object works in the role of a decision engine .

As being mentioned, the Consensus Object needs to analyze data coming from application. Generally, data which comes from the application is the application's goals and the update of application's progress toward those goals. The Application Heartbeat framework that is described in previous section provides the infrastructure for defining and monitoring such information.This means that each application that adopts this framework autonomously sets the goals and updates the progress, while the Consensus Object implements Heart Rate Monitor interface to monitor the target heartbeats. This solution is reasonable since only the application knows what its goals are and when it has finished the computation of a fraction of information.

3.4.2 Communication infrastructure

This subsection aims at providing an experimental view of implementing a self-adaptive software based on analyzing the dataflows between its components.

3.4.2.1 Communication between the Consensus Object and Services

The first task requires the Consensus Object to be aware of available services in the system and their characteristics. There are two methods to discover the availability of services based on the active role of the Consensus Object and they are summarized as follow:

- Pull method: The Consensus Object plays an active role in discovering the availability of services. By saying active role, we mean that the Consensus Object periodically updates its registry about available services. In order to do so, the most simple solution is to define a common location where services can export their information and the Consensus Object can update its knowledge by importing those information. The advantage of this method is that neither service or the Consensus Object needs to know a $pre-information$ about each other. Their statup order also should not be important which mean they can be executed in seperate process with any assumption about the other's existence. However, it is necssary to $hard-code$ in both the Consensus Object and the service a common location. Through this preliminary communication channel the service might inform the Consensus Object of its name, process ID, and all the information

needed to establish a new channel where to exchange control directions. The implemented Consensus Object can periodically scan the common location to update needed information after its startup. The registration mechanism is identical among each service to keep the implementation as modular, expandable, and flexible.

- Push method: The Consensus Object plays an passive role in discovering the availability of services. Utilizing this method means services should have the Consensus Object's ID as a $pre-knowledge$. The Consensus object has to provide an interface for services to register/unregister their availability and other information. After its startup or before being disabled, service use the Consensus Object to inform its availability. It is obviously that push method is not so flexible as pull method as it requires the Consensus object to be started prior to all process but of course it reduces the amount of exchanged information.

The Consensus Object not only stores service's registration but use them to setup a communication channel to control the service. There are many techniques to implement the communication channel using shared memory, file or even interrupt mechanism. Independently from implementation methods, all services are needed to expose a standardized interface so that the Consensus Object can easily take the control by issuing high-level commands and command's target. The commands will decide the interface exposed by each service. Examples of commands and thus service's exposed interface might be:

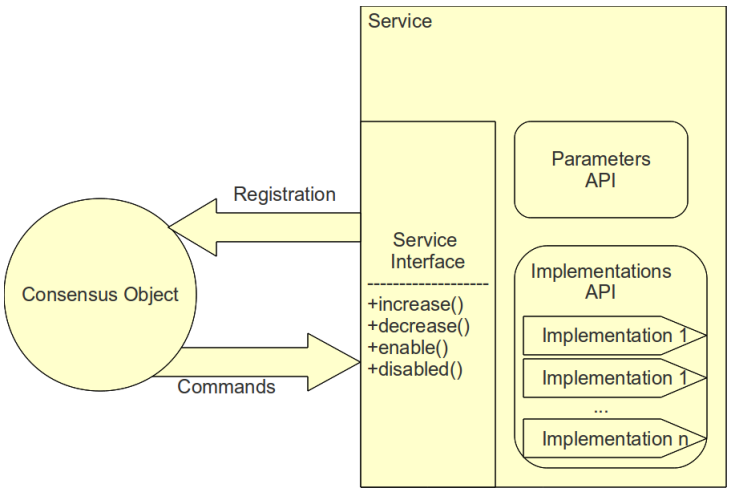

Figure 3.2: The communication between the Consensus Object and Service

- Increase performance.
- Decrease performance.
- Enable service.
- Disable service.

An example of target might be the ID of a process, which has to be affected by the service. the process's ID has to be communicated because the service, in most cases, will be very general and might target different processes in different ways. The Figure 3.2 depicts the information flows between the Consensus Object and services.

3.4.2.2 Communication between the Consensus Object and Applications

The second task requires the Consensus Object to have knowledge about target applications. We have already dicussed that the Application Heartbeats framework is the ideal solution to complete this task. Each target au-

tonomously implement Application Heartbeats framework to define their goals and update the progress towards their goals. Those information is stored in a common place for being retrieved by others who are interested in. This procedure is often called application registration due to the similarity with the service registration procedure described above. At the moment, Application Heartbeats framework provides two model for exposing heartbeats information: one uses shared memory technique while the other uses files. The defined common location because of this could be either a shared memory key or a directory in file system.

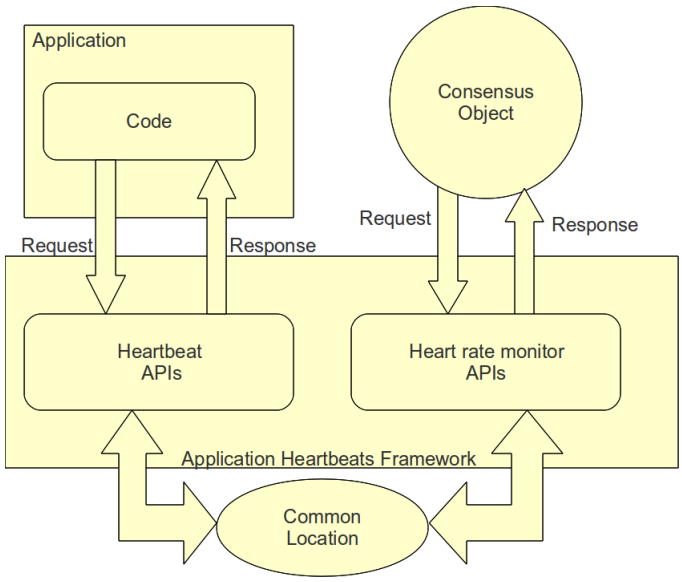

Figure 3.3: The communication between the Consensus Object and Application

The Consensus Object updates information about each application's goals and its progresses by looking into defined common location. For supporting update actions, Application Heartbeats framework also make available a set of API functions called Heart Rate Monitor which enables

the Consensus Object to read the collected data and statistics. The Figure 3.3 depicts what we have just discussed above.

3.4.2.3 Decision support mechanism

The last task requires the Consensus Object to have the ability to analyze data comming from applications and to make decision on which application need to be targeter and which service to activate. This can be done using decision support mechanism. We already disscussed about those mechanisms when we introduced the logics should be implemented by Decide process of ODA loop. The common execution scenario will be summarized as follow:

- The Consensus Object goes through all the available processes and evaluating for each process its progress towards its defined goals.
- According to the set of defined decision policies (either $hard-coded$ or dynamic installed), the Consensus Object knows that whether an this application is $under-performing$ or $over-performing$. All the availble services are scanned to see whether there exists at least one that can be enabled, disabled, or altered to make the application reach closer to its defined goals. If such service exists, the Consensus Object issues appropriate command to it.
- At a higher level, when there exists more than one service which can be controlled, the Consensus Object may implement a mechanism to rank those solutions and chooses the best solution based on its ranking.

3.4.3 Application's goals management

The picture of self-adaptive software implementation has not been completed due to the unknow mechanism based on which services can alter alter their targeted applications. It is a wide range of research and strongly related to what we already disscused previously about the elements that can affect application's goals.In [9], they introduced to complete the picture with the use of `autonomic libraries`. In fact, building autonomic library only concentrates on the first two of listed elements : application's knobs and application's implementations. Thus, two libraries were proposed:

- Parameters API: taking an idea from the fact that each application implements a dynamic algorithm which provides a set of parameters which can be tuned to get better performance during runtime, parameters API contains different sets of parameter with their value. Utilizing parameters api lets service tune the performance of application during runtime by switching between those sets. It is a simple but useful method although it needs human intervention on eliciting sets of parameters to provide a pre-knowledge.

- Implementation Library: refers to a library that exposes some functionalities and has the capability of switching among implementations. This idea comes from the fact that some applications provide more than one implementation for their logic. Each algorithm has its own advantage in specific context and it is desirable to have the library automatically chooses the best one based on current context.

It's obviously that a high abstraction of implementations which provide the same functionality should be provide so that application should never see the switching. This means each implementation implements the identical interface and working on a unique abstract data (e.g., a translation between abstract data and real data used by the algorithm may be needed). Moreover, for a correct choice of the implementation, the library must have data regarding goals and a decision support mechanism.

The Figure 3.4 depicts the usage of parameters api library while the Figure 3.5 depict the method of using implementations library in building self-adaptive software.

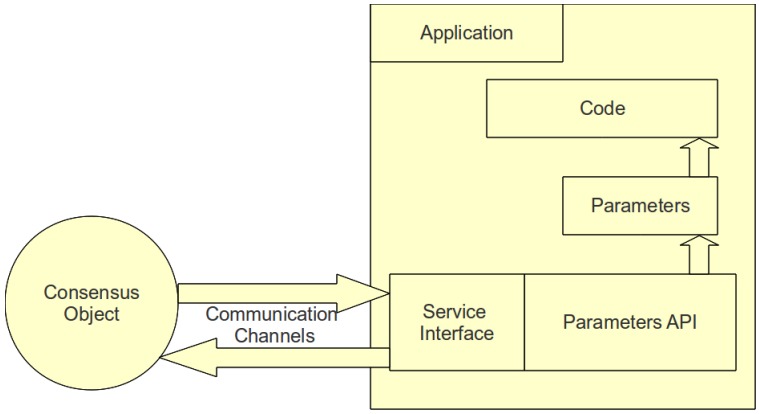

Figure 3.4: The Parameters API

3.5 Summarization

In this Chapter, we have already disscused basic technical methods for building a self-adaptive software system and their connection. We also in-

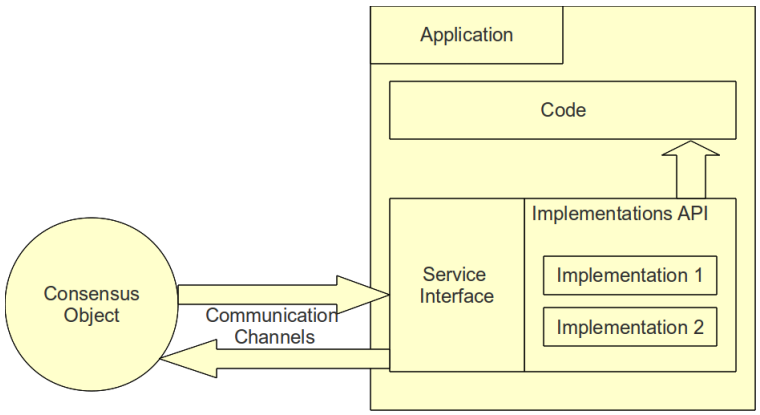

Figure 3.5: The implementations library

troduced Application Hearbeats framework which is very useful in providing an infrastructure for defining, and monitoring application's goals and its progress towards goals. All of those knowledges will be illustrated in the next chapter while we step by step design and build a specific self-adaptive software system.

Chapter 4

Implementation

When the definition of self-adaptive software systems is introduced, many people think about very complicated software systems but in fact self-adaptive software systems has a spread spectrum, ranging from great distributed systems to very simple software applications run on our mobile phone. At the beginning of this Chapter, we are going to introduce a mobile operating system called Android. We then describe the design and implementation of a self-adaptive MP3 decoder as an illustrative example. This application takes a music MP3 file as input and generate PCM (Pulse code module) samples as output so that they can be played by audio device. During this Chapter, we are going to illustrate most of the methods and solutions that we have already discussed in previous chapter.

4.1 Android Operating System

4.1.1 Introduction

The term `Android` has its origin in the Greek as word $andr-$, meaning "man or male" and the suffix $-eides$, used to mean "alike or of the species". This together means as much as "being human". Android is a software stack for mobile devices that includes an operating system, middleware and key applications. Android was initially developed by Android Incorporation. Later it is acquired by Google to form `Open Headset Alliance` which is a group of 65 hardware, software, and telecom companies. Not like other mobile operating system such as Microsoft Mobile, Symbian or Iphone, Android is an open source project which means everybody can contributed to its development. By saying open we mean that :

- From industrial point of view : the software stack is a open source project under the Apache 2.0 license and is available after first handsets ship. Anyone can download and build the system image for their own device.

- From user's point of view : users have control of their experience and they control what gets installed.

- From developer's point of view : Developers do not need permission to ship an application. The framework APIs are made open without any privileges and developers can integrate, extend or replace existing components.

Android group also provides a Software Developement Kit (SDK) along with the OS to help developers who want to develope applications for Android system without knowing the underlying complications. Android applications are developed using Java language. Each application will run in different virtual machine. Although developer use Java language for developing applications, the virtual machine in which they are executed is not Java Virtual Machine (JVM) but Dalvik Virtual Machine. Dalvik Virtual Machine is also an open source project and gives a better performance than general JVM in the area of mobile systems. It is the open property of Android which allow developer to access to the platform's low levels influenced us in choosing Android system as the developing enviroment for our example of self-adaptive software.

4.1.2 Android system architecture

The Figure 4.1[1] depicts the architecture of Android operating system. The software stack is devided into four layers and will be summarized as follow:

- Application layer: The Android software platform will come with a set of basic applications like browser, $e-mail$ client, SMS program, maps, calendar, contacts and many more. All these applications are written using the Java programming language. It should be mentioned that applications can be run simultaneously, it is possible to hear music and read an $e-mail$ at the same time. This layer will

[1]Graphic taken from website http://developer.android.com/guide/basics/what-is-android.html

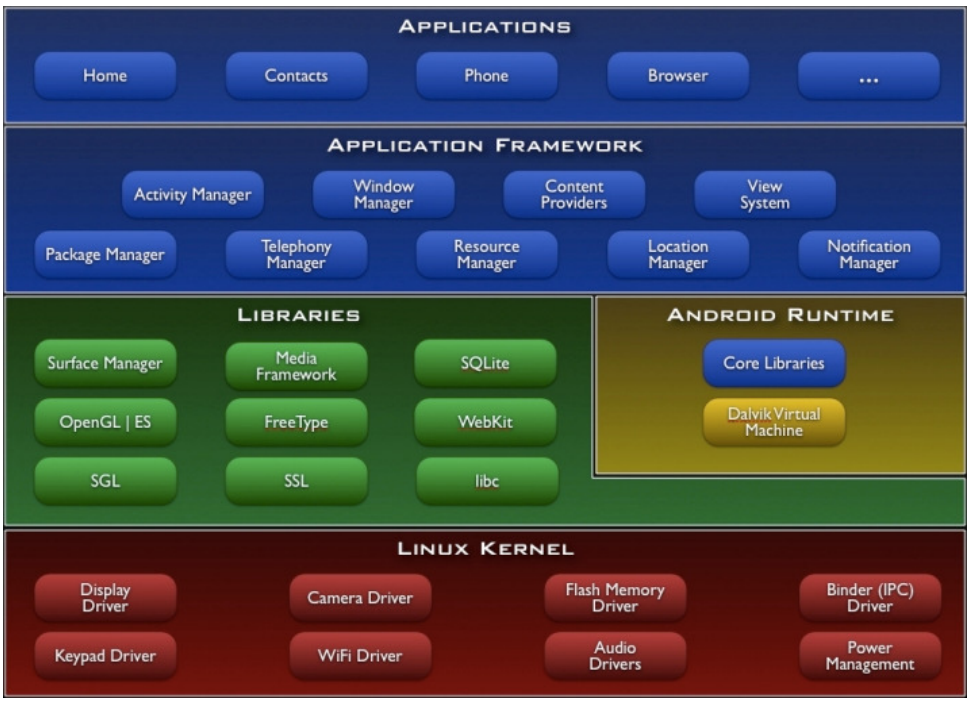

Figure 4.1: Android operating system architecture

mostly be used by commonly cell phone users.

- Application framework layer: An application framework is a software framework that is used to implement a standard structure of an application for a specific operating system. With the help of managers, content providers, and other services, it can reassemble functions used by other existing applications.

- The libraries: Android includes a set of C/C++ libraries used by various components of the Android system. These capabilities are exposed to developers through the Android application framework. They will be called through a Java interface. Some of the core libraries are listed below:

  - System C library: a BSD-derived implementation of the standard C system library (libc), tuned for embedded Linux-based devices
  - Media Libraries: based on PacketVideo's OpenCORE; the libraries support playback and recording of many popular audio and video formats, as well as static image files, including MPEG4, H.264, MP3, AAC, AMR, JPG, and PNG.
  - Surface Manager: manages access to the display subsystem and seamlessly composites 2D and 3D graphic layers from multiple applications
  - ibWebCore: a modern web browser engine which powers both the Android browser and an embeddable web view
  - SGL: the underlying 2D graphics engine.
  - 3D libraries : an implementation based on OpenGL ES 1.0 APIs; the libraries use either hardware 3D acceleration (where available) or the included, highly optimized 3D software rasterizer.
  - FreeType:bitmap and vector font rendering.
  - SQLite:a powerful and lightweight relational database engine available to all applications.

- Android Runtime: The Android runtime consists of two components. First a set of core libraries which provides most of the functionality available in the core libraries of the Java programming language. Second the virtual machine Dalvik which operates like a translator between the application side and the operating system. Every applica-

tion which runs on Android is written in Java. As the operating system is not able to understand this programming language directly, the Java programs will be translated by the virtual machine Dalvik. The translated code can then be executed by the operating system. Unlike JVMs, which are stack machines, the Dalvik Virtual Machine is a register-based architecture. Other very important notice is that applications will be encapsulated in Dalvik Virtual Machine. For every program an own virtual machine is available even if some programs are running parallel. The advantage is that the different programs do not affect each other, so a program error for example can lead to a crash of the program but not of the whole system.

- Linux Kernel Layer: Android is built on top of a solid and proven foundation: the Linux kernel. Linux provides the hardware abstraction layer for Android, allowing Android to be ported to a wide variety of platforms in the future. Internally, Android uses Linux for its memory management, process management, networking, and other operating system services. The Android phone user will never see Linux, and programs will not make Linux calls directly.

4.1.3 Working with the Android platform

The implementation of our self-adaptive software needs to utilize the Application Heartbeats Framework which is written in C/C++ language. As being introduced, the Software Developement Kit only allows developers to write application and library in Java language. Thus, there is two

methods to enable the Application Heartbeats framework to work in Android are : either we port the Application Heartbeats framework's source code into Java language or implement our self-adaptive software at low layer of the software stack rather than in Application layer. The second solution seems to be the most reasonable one because the success in implementing our software application at low level can be expanded to provide adaptivity at general level. Moreover, because Android is built on top of Linux kernel, most of its low level library are written in C/C++, thus there is a posibility of utilizing the Application Heartbeats framework without porting requirement. By saying low level, we mean Application Heartbeats framework will become part of Android's libraries working in the layer right upper the Linux kernel. Therefore, our MP3 decoder application will be part of the Application Framework layer.

This solution requires us to download Android platform source code, and have knowledge of building system image. Downloading and setting up the enviroment for building Android platform has been described on Android official website [2] or from my WordPress article [3]. The system image can be build using `make` command at source code's top level directory. The `make` command supports a lot of paramters for building the system image suites developer's need (e.g., device's architecture, device's hardware configuration...). It is notable point that, Android build system does not use the normal `Makefile` but uses its own `Makefile` called `Android.mk`. Contributions to Android base framework are now left with the coding

[2] Android Platform http://source.android.com/source/index.html
[3] How to download and build Android platform - http://thanhpt25.wordpress.com/

phase and creating appropriate `Android.mk` file. The snip code 4.1 depicts an example `Android.mk` used in our project to build Application Heartbeats frameworks as a shared libraries in Android.

```
1  LOCAL_PATH:= $(call my-dir)
2  include $(CLEAR_VARS)
3
4  LOCAL_ARM_MODE:=arm
5  LOCAL_C_INCLUDES:=\
6        $(call include-path-for)
7
8  LOCAL_SRC_FILES:= \
9        heartbeat-file.cpp
10 LOCAL_PRELINK_MODULE:=false
11 LOCAL_MODULE:=libhb-file
12 include $(BUILD_SHARED_LIBRARY)
```

List 4.1: A sample Android.mk file

The most important parts of an Android.mk file can be illustrated as bellow:

- An Android.mk file must begin with the definition of the LOCAL_PATH variable. It is used to locate source files in the development tree. In this example, the macro function $my - dir$, provided by the build system, is used to return the path of the current directory (i.e. the directory containing the Android.mk file itself).

- The CLEAR_VARS variable is provided by the build system and points to a special GNU Makefile that will clear many LOCAL_XXX variables (e.g. LOCAL_MODULE, LOCAL_SRC_FILES,etc.), with the exception of LOCAL_PATH. This is needed because all build control

files are parsed in a single GNU Make execution context where all variables are global.

- The LOCAL_MODULE variable must be defined to identify each module we describe in our Android.mk. The name must be *unique* and not contain any spaces. The build system will automatically add proper prefix and suffix to the corresponding generated file.

- The LOCAL_SRC_FILES variables must contain a list of C and/or C++ source files that will be built and assembled into a module. Note that we should not list header and included files here, because the build system will compute dependencies automatically for us.

- The last line indicates the type of our project (e.g., static library, shared library or $stand - a - lone$ application).

4.2 Self-adaptive MP3 decoder implementation

This section represents to readers the implementation of a self-adaptive MP3 decoder. During the implementation process, we are going to try to follow the methodology which was dicussed in the Chapter 3.

4.2.1 General description

As being represented to readers, self-adaptive software systems are born to solve the problem of skyrocketting complexity in current software systems. Therefore, we concluded that a self-adaptive software system should not add more complexity to the design. That is why we chosed to develope

a very small self-adaptive MP3 decoder which works on a mobile operting system - Android to illustrate the benefit of self-adaptivity. The Figure 4.2 represents to readers the overview of the system we are going to develope.

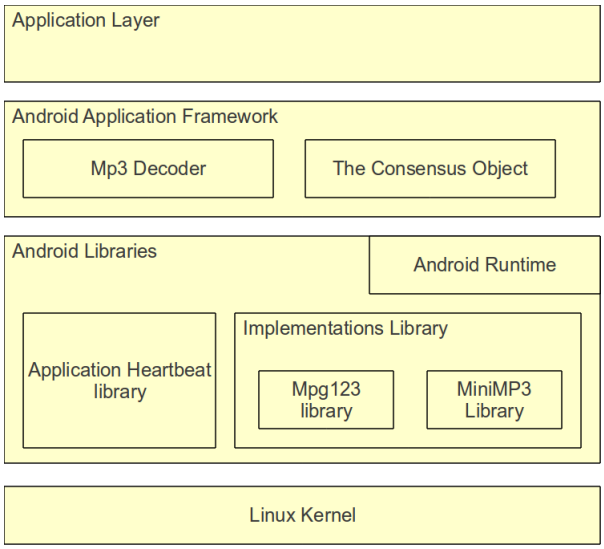

Figure 4.2: The self-adaptive MP3 decoder system.

As can be observed, the Application Heartbeats Framework will be ported to be part of the Android's core libraries. Whereas, our self-adaptive MP3 decoder and its adaptation engine - the Consensus Object are part of the Android Application Framework. We also developed an implementations library contains two algorithms for decoding MP3 data into PCM samples: the MPG123 algorithm and the MiniMP3 algorithm. It is obviously that for flexibility and portability, the implementations should also be a part of the Android's core libraries.

From the internal approach point of view, the system has the ability to gather the information about its state and its operation context to decide at runtime which configuration among the set of its available running con-

figurations will be chosen for the next operation according to the different criteria. Here we proposed the definion of running configuration instead of specific algorithm implementation. System's running configuration has a broader meaning than implementation in the sense that it includes the algorithm implementation and a set of system paramter's values. Different running configuration may use the same algorithm implementation but different set of system parameter's values provides the ability to unify the usage of parameters API and implementations library. Moreover, as we dicussed about adaptation engine, we agreed that we need a mechanism to evaluate the potential of each solution so that the decision making engine can choose the best among them. We also concluded that for simplicity, those evaluations can be pre-evaluated according to some criterias using human knowledge. Thus, by introducing running configurations we can make them to contain also their priority of potential to help decision making engine to make a better chose. In fact, evaluation a solution is a complicated process which not only based on the solution itself but also based on the situation it is applied to. This problem relates to the area of context definition and action definition for each specific context which are out of the scope of this document.

From the external approach point of view, there exists an adaptation engine called the Consensus Object which monitors the progress towards application's defined goals and issues appropriate commands to swith between different available configurations to save the situation.

The purpose the system is to provide an acceptable quality of music

to listener. On mobile device, many factors can effect this purpose such as the low bandwidth, the defect of current used algorithm on new hardware configuration, etc. To accomplish the purpose, the system must have the ability to switch between different configuration to adapt the new situation actively or passively.

4.2.2 Porting Application Heartbeats Framework

Thank to the portability of Application Heartbeats framework, it is quite simple to port the framework into Android platform. As dicussed in Chapter 3, Application Heartbeats framework provides more than one model for implementing the communication methods: the shared memory model and file model. However, shared memory has been stripped off from Android platform because Android development team considered it as un-safe operation. Because of this, we were left with only one choice of using the file model. There are two steps in porting Application Heartbeats framework as an Android's core shared library:

- The source code will be located in a specific location so that it can be found by build system.
- An appropriate `Android.mk` need to be created and located in the same folder with framework's source code. The snip code 4.2 depicts the `Android.mk` file which was created and used in our project.

```
1 LOCAL_PATH:= $(call my-dir)
2 include $(CLEAR_VARS)
3
4 ifneq ($(TARGET_SIMULATOR),true)
```

```
5  LOCAL_SHARED_LIBRARIES += libdl
6  endif
7
8  LOCAL_ARM_MODE := arm
9  LOCAL_C_INCLUDES := \
10         $(call include-path-for)
11
12 LOCAL_SRC_FILES  := heartbeat-file.cpp
13 LOCAL_PRELINK_MODULE := false
14 LOCAL_MODULE := libhb-file
15 include $(BUILD_SHARED_LIBRARY)
16
17 include $(CLEAR_VARS)
18
19 ifneq ($(TARGET_SIMULATOR),true)
20       LOCAL_SHARED_LIBRARIES += libdl
21 endif
22
23 LOCAL_ARM_MODE := arm
24 LOCAL_C_INCLUDES := \
25       $(call include-path-for)
26
27 LOCAL_SRC_FILES  := \
28       heart_rate_monitor-file.cpp
29 LOCAL_PRELINK_MODULE := false
30 LOCAL_MODULE := libhrm-file
31
32 include $(BUILD_SHARED_LIBRARY)
```

List 4.2: Android.mk file for building Application Heartbeats

As the result, two libraries named libhrm-file and libhb-file will be created in our Android system image. Any application or library which is interested in invoking Application Heartbeats APIs should link to those two

library as local shared library.

4.2.3 The implementation of adaptable MP3 decoder

We dicussed in Chapter 3 about how Application Heartbeats framework provides a flexibility way for application in goal definition and monitoring their progress towards goal. For a specific application, it is first necessary to define what application's goals are. It is obviously that an MP3 decoder should provide a stable data stream for the audio decoder so that users can enjoy music without any interruption. Thus, it will be useful if a heartbeat is generated for every block being decoded, while the key metric here is likely the frequency of heartbeat or heart rate. The Figure 4.3 represents the sequence diagram of an MP3 decoder using Application Heartbeats framework to provide performance information.

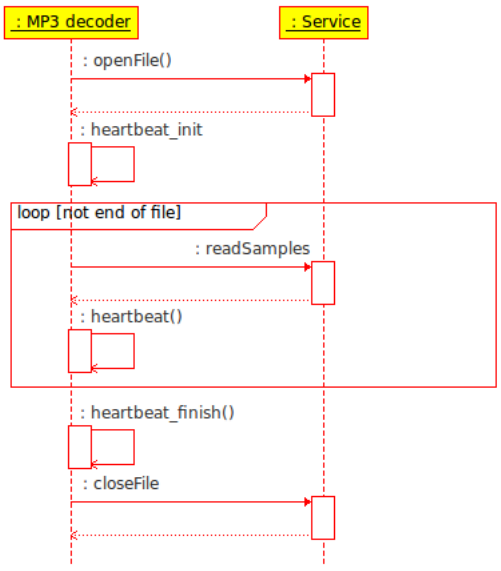

Figure 4.3: Registering performance using Application Heartbeats library

As can be observed, our MP3 application first does the framework initialization by calling heartbeat_init() API. This operation will request the framework to created a file in specified common place to store application's heatbeat information. It then requests the implementation service to open a specific MP3 file for being decoded. After each successfully decoded data block, MP3 decoder application issues a heartbeat to register the application's progress. This progress information can be monitored and analyzed by application itself or by external applications.

In case we have self-optimized application software , our MP3 decoder needs to monitor its own progress towards goal, analyzes and takes appropriate action. We have already mentioned in table 3.1 that Application Heartbeats framework also provides APIs for application to monitor the its own heart rate (e.g., hb_get_global_rate and hb_get_windowed_rate). It is straightforward that we can make use of those APIs with the definition of desired heart rate to enable our application to realize the situation and make appropriate action. The Figure 4.4 represents how heart rate related APIs can be employed in our project.

The application retrieves its own windowed heart rate after each iteration and compares the windowed heart rate with the maximum and minumum target heart rate. A very simple decision tree in the form of if-then-else structure is employed here to help making decision: if the windowed heart rate is greater than the acceptable maximum heart rate or less than minimum acceptable heart rate then the application will issues the implementation switching service to change the running configura-

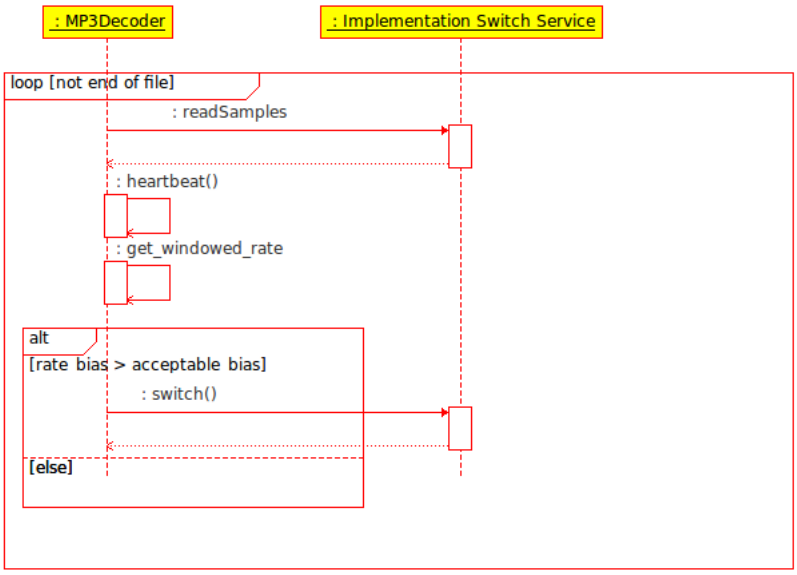

Figure 4.4: Sequence diagram for self-optimization MP3 decoder

tion; otherwise it continues the next iteration. The desired heart rate is a pre-knowledge and is specified when we call heartbeat_init(). Application Heartbeats Framework allows developers to specify the maximum heart rate and minimum heart rate as upper bound and lower bound respectively for target heart rate during initialization. This ability provides the flexible way for defining the target goal because application can define a acceptable interval instead of a specific value. The snip code 4.3 used in our project is represented below with comments so that it can be straightforward to understand:

```
1 //Bias is the pre-specified value
2 heartbeat_init(&heart,
3          minTargetRate,
4          maxTargetRate,
```

```
5           window_size, buffer_depth,
6           log_name);
7   ...
8   int tag = getppid (); //Tag indicator
9   int iters = 0;
10  //Here we start our main loop of decoding
11  do{
12    iters ++;
13    if(hbEnabled)
14       //Update progress towards goal
15       heartbeat(&heart, tag);
16       //Decoding MP3
17       int nSamples = 0;
18       nSamples = readSamples(
19             handle,
20             outputBuffer,
21             OUTPUT_BUFFER_SIZE);
22       if (nSamples ==0 )
23           break;
24       else{
25           //Play our decoded samples
26           lpTrack->write(outputBuffer,
27              nSamples * sizeof(unsigned char));
28       }
29       /*Seft-awareness and adaptation*/
30       if (hbEnabled){
31          float globalRate   = 0.0;
32          float windowedRate = 0.0;
33          //Retrieve the rate
34           globalRate   =
35                hb_get_global_rate(&heart);
36           windowedRate =
37                hb_get_windowed_rate(&heart);
38           float maxRate=
39                hb_get_max_rate(&heart);
```

```
40          float minRate=
41               hb_get_min_rate(&heart);
42          if((windowedRate > maxRate) ||
43              //Over-performance
44              //switch to lower potential config
45              swithConfig(false);
46          }else if  (windowedRate < minRate)){
47              //Under-performance
48              //switch to higher potential config
49              swithConfig(true);
50          }else{
51              //Normal execution
52              continue;
53          }
54      }
55  }while(1);
```

List 4.3: Code for self-optimization implementation

4.2.4 The implementation of Services

The logic should be implemented by Service was represented to readers in Chapter 3. In actual implementation, the relationship between service is either one of two types or a mix:

- A. One application and multiple services.
- B. Multiple applications and multiple services.

Other scenarios such as one application and one service or multiple applications and one service are special cases of A and B. In fact, the relationship between applications and services do not affects directly to the

application's progress towards goals. More general, this relationship will be considered by the adaptation engine when it has to evaluate the impact of each decision. In the scope of our project, we propose to implement the most simple relationship between application and its service : One application and one service. This consideration will lead to the fact that application's process ID will be service's process ID. One may argue that there is no meaning in this implementation. However, in our point of view, this implementation not only maintains the overal architecture of self-adaptive software system but also give the simplicity to the system.

Because the relationship is now a one-to-one type, registering an application with the Consensus Object also means registering a service with the Consensus Object. It is true because all service are identical to the Consensus Object except for their identities. We are going to represent to readers how the Consensus Object makes use of this one-to-one relationship in the following section.

We argued that service is an extension of actuator. By saying an extension we mean that it not only represents a component capable of performing changes on one or more applications or on the system but it also includes the interface to interact with the Consensus Object and with the application. Thus, designing and implementing a service is actually designing and implementing those programming interfaces.

- Interface between service and the Consensus Object: We agreed that services should implement the identical interface so that the Consensus Object can command them with the same set of commands. The

Consensus Object distinguishes services by their identity. The only prolem left is how to implement the communication channel which is in one way allow service to register itself with the Consensus Object, while in the reverse direction allows service to receive the commands from the Consensus Object. With one-to-one relationship between application and service, we argue that there is no need to register service to the Consensus Object, because the application registration already means service registration.Thus, the on-going channel of bi-direction communication can makes use of Application Heartbeats framework as well. For the in-coming channel, many methods have been proposed using : file model, pipeline, shared memory, etc. For our simple interface, we propose to utilize software interupt as the communication channel. In our point of view, service will be implemented as a software interupt handler which catchs and handles the following interupts:

- SIGUSR1: When service receives SIGUSER1 from the Consensus Object, it will consider as if it receives a request to adapt the under-performance situation.
- SIGUSR2: When service receives SIGUSER2 from the Consensus Object, it will consider as if it receives a request to adapt the over-performance situation.
- SIGINT/SIGTERM : When service receives SIGINT/SIGTERM from the Consensus Object, it will consider as if it receives a request to disable itself.

– SIGCON :When service receives SIGCON from the Consensus Object, it will consider as if it receives a request to enable itself.

The snip code 4.4 depicts part of our service's implementation as a interupt handler.

```
1  //Up command definition
2  void cmdUp (int sig)
3  {
4     if (currentRunConfig < MAX_N_CONFIGS){
5       currentRunConfig++;
6     }else{
7       currentRunConfig = MAX_N_CONFIGS;
8     }
9  }
10 //Down command definition
11 void cmdDown(int sig)
12 {
13    if (currentRunConfig > 0){
14       currentRunConfig--;
15    }else{
16       currentRunConfig = 0;
17    }
18 }
19 //Start service; register the interface
20 int service_start(){
21    int rc = 0;
22    struct sigaction actionUp, actionDown;
23    sigset_t block_mask;
24    pid_t child_id;
25    /* A SIG_USR1 indicate UP command */
26    sigfillset (&block_mask);
27    actionUp.sa_handler = cmdUp;
28    actionUp.sa_mask = block_mask;
```

```
29    actionUp.sa_flags = 0;
30    rc = sigaction (SIGUSR1,
31                    &actionUp, NULL);
32    if(rc != 0)
33       return rc;
34
35    /* A SIG_USR2 indicate DOWN command */
36    sigfillset (&block_mask);
37    actionDown.sa_handler = cmdDown;
38    actionDown.sa_mask = block_mask;
39    actionDown.sa_flags = 0;
40    rc = sigaction (SIGUSR2,
41                   &actionDown, NULL);
42    if(rc != 0)
43       return rc;
44    ....
45 }
```

List 4.4: Service API implementation

- The autonomic libraries: We have discussed about autonomic library and how service affects system's performance using those libraries. While implementing autonomic libraries, we concluded that it is possible to define the running configuration as a more general concept of algorithm implementation. A running configuration describes the action service should enact including algorithm implementation, a set of parameter's values and a value indicate its level of potential. Service now not only has the ability to switch between different implementations but also has the ability to use the same algorithm implementation but different in parameter's values. Service also can

take advantage of human knowledge in form of configuration's predefined level of potential. The snip code 4.5 depicts the a general structure of configuration.

```
1  enum Methods{
2     LIBMAD,
3     LIBMPG123,
4     LIBMINIMP3
5  };
6  typedef struct{
7     //parameters definition
8     int64_t buffer_size;
9     bool    skip_samples;
10    ...
11    //current used implementation
12    Methods currentMethod;
13 } Configuration;
```

List 4.5: Sample structure of a configuration

We developed an implementations library which contains three MP3 decoder implementations:

- MAD implementation : MAD uses integer computation rather than floating point, it is well suited for architectures without a floating point unit. All calculations are performed with a 32-bit fixed-point integer representation.
- MPG123 implementation : Unlike MAD using float for decoding so it perfectly works well on architecture that possesses a Floating Point Unit (FPU). There is a downgrade when MPG123 has to work on architecture which does not support FPU.

– MiniMP3 implementation : Both MAD and MPG123 uses assembly code for speeding up decoding. MiniMP3 does not use assembly code, therefore it has the slowest decoding speed.

We have already argued that to develope an implementations library, a high abstraction of input and output data used by algorithms must be designed so that application could not aware of the switching behind the scene. Thus, impementations library also need to have the ability to translate from the abstracted input data type to specific input data type used by current algorithm, and from output data type returned by current algorithm to abstracted output data type understandably to application. In our project, we provided those abstraction by utilizing two methods: (1) Abstracted structures: contain all information needed by each algorithm even if they are redundances. (2) Translate function : Translate abstracted structure into structure used by algorithm.

It is also notable that a quiescent state needs to be reached before we can change the underlying implementation. In case of MP3 decoder, a quiescent state could be the state when all decoded data by one implementation has been transfered to playback device. At that state, the implementations library can switch to another implementation without affecting application progress. The snip code 4.6 illustrates the logic implemented in our MP3 decoder to reach the quiescent state.

```
1 if ((lastMethod == LIBMPG123) &&
2     (currentMethod == LIBMINIMP3))
```

```
3  {
4      //Not yet reached quiescent state
5      if (mp3Handle->leftSamples>0)
6      {
7          //continue using old method
8          return mpg123_readSamples(index,
9              buffer,
10             mp3Handle->leftSamples);
11     }
12     else
13     {
14         //reached quiescent
15         //can change method now.
16         lastMethod = currentMethod;
17         return minimp3_readSamples(
18                                 index,
19                                 buffer,
20                                 size);
21 }
22 }
```

List 4.6: Code for implementation switching

4.2.5 The implementation of the Consensus Object

As being introduced in Chapter 3, the Consensus Object plays a center role of the control loop. We also argued that the Consensus Object should be implemented to accomplish the following tasks:

- Discovers and dynamically updates availability and characteristics of services.
- Discovers and dynamically updates the possible targets aimed at by

the services.

- Analyzes data coming from the applications, decides which applications need to be targeted, chooses which service to activate or to modify, and submits the plans to the services.

The first task requires the Consensus Object to update the availability and characteristics of services. Many techniques have been dicussed to build the communication channel between services and the Consensus Object (e.g., shared memory, pipeline, file model,etc.). The purpose of all those methods is to let the Consensus Object aware of available services' process IDs so that it can issues commands in adapting phase. In our project, to simplify the implementation, a strong coupled one-to-one relationship between application and its service is implied. This implication allow the Consensus Object to by pass the discovering process of available services because while taking the discovery process looking for target applications, the Consensus Object will know their process ID. The one-to-one relationship between application and its service allows the Consensus Object to make the implication about service's process ID as well. Therefore, the implementation logic of the Consensus Object will be summarized as follow:

- Discovers and dynamically updates $heartbeat - enabled$ application. The process ID of application and its service are now the same because of one-to-one relationship.

- Analyzes data coming from the applications, decides which applications need to be targeted, makes decision, and submits the plans to

the services.

The first task can be accomplished by scanning in the defined common place, because each heartbeat-enabled application will create a file with its process ID as file name (using Application Heartbeats Framework's file model). The snip code 4.7 depicts this scanning process.

```
1  DIR *dp;
2  int count = 0;
3  struct dirent *ep;
4  dp = opendir (
5       getenv("HEARTBEAT_ENABLED_DIR"));
6  if (dp != NULL)
7  {
8      do{
9        ep = readdir (dp);
10       if(ep!=NULL){
11          int id = atoi(ep->d_name);
12          if(id > 0){
13             listIds[count] = id;
14             count++;
15          }
16       }
17     }while(ep!=NULL);
18     closedir (dp);
19 }else{
20     return -1;
21 }
22 return count;
```

List 4.7: Code for target application discovery

After scanning, the Consensus Object maintains a list of process IDs. The second task requires the Consensus Object to have the ability to re-

trieve target's heartbeats information, to analyze and to make decision. Retrieving target's heartbeats information process can be accomplished using Application Heartbeats framework APIs. In Table 3.2, we represented a list of APIs that framework provides to external application. We alsp propose to store targets' information in a well-formatted structure called Registry table. Each registry table's element contains heartbeats information of one target. The Figure 4.8 shows the structure of registry table and registry element.

```
1  typedef struct{
2    pid_t pId;
3    heart_rate_monitor_t hrm;
4    heart_data_t* records;
5    int nRecords;
6  } RegistryElement;
7
8  typedef struct{
9    int size;
10   RegistryElement* regs;
11 } Registry;
```

List 4.8: Sample of the Consensus Object's registry table

The Consensus Object will initialize one heart rate monitor for each process in its list. Those monitors provide a handle to access target application's progress towards goal information. By iterating through the list of process and retrieving information using handle, a registry table will be built contains heartbeat data records belong to each process. The snip code 4.9 represent the logic which takes reponsibility to update the registry table.

```
for(i = 0; i < nIds; i++){
  //Init monitor for target app
  int rc = heart_rate_monitor_init(
             &(regs[i].hrm),
               regs[i].pId);
  if(rc < 0){
     return rc;
  }
  heart_data_t* records = (heart_data_t*)
  malloc( MAX * sizeof(heart_data_t));
  regs[i].records = records;
  int current_tag = -1,
      last_tag = -1;
  int j = 0;
  while(last_tag < MAX-1) {
     heartbeat_record_t record;
     //Update heart record data
     while(current_tag == last_tag) {
        int rc = -1;
        while (rc != 0)
           rc = hrm_get_current(&(regs[i].hrm),
                                &record);
        current_tag = record.tag;
     }
     records[j].tag =
              last_tag = current_tag;
     records[j].global_rate=
                record.global_rate;
     records[j].min_rate=
            hrm_get_min_rate(&(regs[i].hrm));
     records[j].max_rate=
            hrm_get_max_rate(&(regs[i].hrm));
     records[j].window_rate=
            record.window_rate;
```

```
35      j++;
36    }
37    regs[i].nRecords = j;
38    //Finish monitoring
39    heart_rate_monitor_finish(&(regs[i].hrm));
40  }
```

List 4.9: Code for updating registry table

This registry table is going to be used as the input parameter for decision engine which takes part in analyzing the data and providing decision as output. We employed a same decision logic dicussed in previous section for application's self-adaptation. The Figure 4.10 depicts the logic used by the Consensus Object to analyze data and to make decision:

```
1  //Get the list of registry element
2  RegistryElement* regs = registry->regs;
3  if(regs == NULL){
4     rc = -1;
5     return rc;
6  }
7  int i = 0;
8  //Start analyzing data
9  for (i = 0; i < nIds; i ++){
10     pid_t pId = regs[i].pId;
11     heart_data_t record =
12           regs[i].records[regs[i].nRecords];
13     double window_rate = record.window_rate;
14     if (window_rate > maxRate)
15     //over performance -> Issue Down command
16         kill(pId, SIGUSR2);
17     else if (window_rate < minRate)
18     //under performance -> Issue Up command
19        kill(pId, SIGUSR1);
```

```
20      else
21          continue;
22  }
```

List 4.10: Logic for processing registry table

One notable point here is the employ of interupt signal. The Consensus Object as can be observed use `kill` system call with parameters are service process ID and signal type to issue command to service. In the section discussed about the implementation of service, we argued that our simple service was implemented as a interupt handler, which catch the user defined signal SIGUSR1 and SIGUSR2 and to provide the interface with only 2 command : increase performance and decrease performance. Here by introducing `kill` system call we completed the picture of communication channel between the Consensus Object and the service.

Those processes represented to readers about are main elements of the main logic loop. The Consensus Object periodically executes the loop to update information and to make decisions. By implementing this loop, the Consensus Object plays the role of an adaptation engine, enables the adaptivity to target applications.

4.3 Summarization

In this Chapter, we first introduced to readers about the Android mobile operating system as our project's developing enviroment. The reason we chose Android is to illustrate the spectrum of self-adaptivity benefit. Moreover, Android is a open source project which allow developer to ar-

bitrary change and build the system image for their own purpose without permission. Thus, this provides us a very flexible developing enviroment. We then represented to readers our project work as an illustrative example in building a self-adaptive software system. This Chapter has a strong relationship with Chapter 3 because during implementation process, we employed as many as possible those technical methods discussed in Chapter 3. We also provided idea about the concept of running configuration as a unification of Parameters API and Implementations Library.

Chapter 5

Experimental result

In this Chapter, we are going to represent the experimental result of running our project to verify the benefit of self-adaptive property. It is obviously that using Application Heartbeats framework add overhead to application. Therefore, we first analyze the effect of Application Heartbeats framework on execution time. We then simulate the conditions that affect the decoding speed (e.g, low band-width, architecture does not support FPU, etc) to see how our software application adapts itselft to the new situation. At the end of this Chapter, we will represent a summarization based on actual experimental results.

5.1 Application Heartbeats framework overhead

5.1.1 Test purpose and expected result

The purpose of this test is to verify the overhead effect of using Application Heartbeats framework. It is oviously that, the framework adds

overhead to application because it uses a series of system calls to initialize data structures and to store heartbeats information. As using file model of the framework, read/write to file operations may cause notable impact on application execution time. Therefore, our expectation is to verify that the overhead is acceptatble.

5.1.2 Testing conditions

The experiment was undertaken in generic condition:

- Software condition: A generic build of Android System image was built for testing purpose.
- Hardware condition: Instead of using real device, our tests was undertaken on emulator provided along with the SDK. The main hardware configuration options are :
  - Processor: ARMv7 CPU.
  - Cache's size: 66Mb.
  - RAM's size: 96Mb.
  - Max Virtual Machine application heap size: 16Mb.
  - SD Card : No.
  - Internal Storage : 32Mb.
  - Audio Plackback support : Yes.
  - Cache partition: yes.

5.1.3 Experimental Result and Comments

We executed the test by logging the interval needed to complete decoding each MP3 file with and without Application Heartbeats being enabled. We chose to decode the whole file instead of each decoded data block because our algorithms are implemented as a stream decoder. As a stream decoder, the interval needed to decode a frame depends on whether that frame is already decoded and put in buffer or it has to be read from a stream to be decoded. The later obviously takes more time. We first tested Application Heartbeats framework's overhead with playback disabled to remove the effect of playback time. The table 5.1 depicts the experimental results executed on nine differents input files.

Table 5.1: Application Heartbeats overhead no playback

	MPG123			MiniMP3		
	With HBs(us)	W/o HBs	Impact(%)	With HBs(us)	W/o HBs	Impact(%)
2.1Mb	12524	10196	22.8	16538	15255	07.7
3.3Mb	17763	14906	19.1	24679	22460	08.9
4.3Mb	23113	19055	21.2	33370	29717	10.9
5.3Mb	28053	22262	26.0	40471	34453	14.8
6.4Mb	28047	23220	20.7	39490	34824	11.8
7.3Mb	17946	15050	19.2	23721	21011	11.4
8.2Mb	19884	16606	19.7	26029	23181	10.9
9.9Mb	23324	19790	17.8	31812	29850	06.2
12.6Mb	31650	25925	22.0	40009	37738	05.6

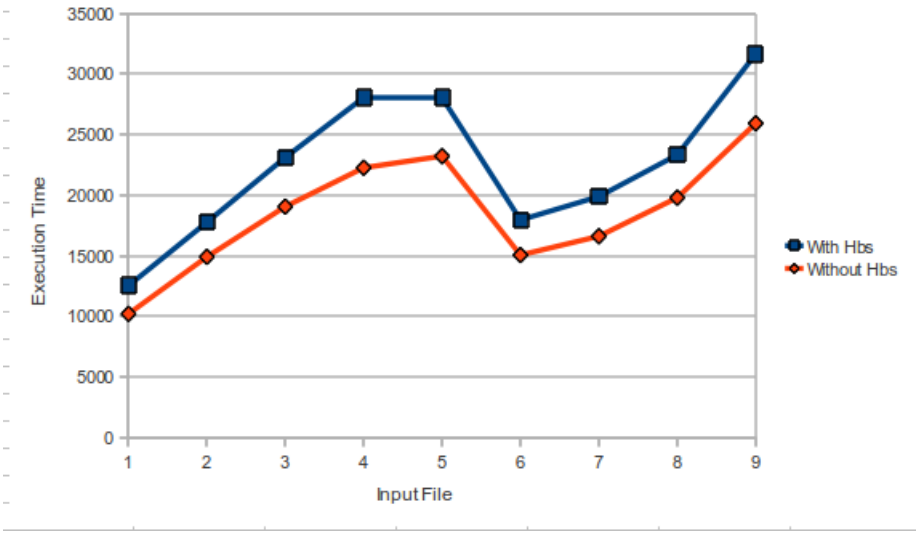

Figure 5.1: Overhead on application execution time - MPG123 decoder

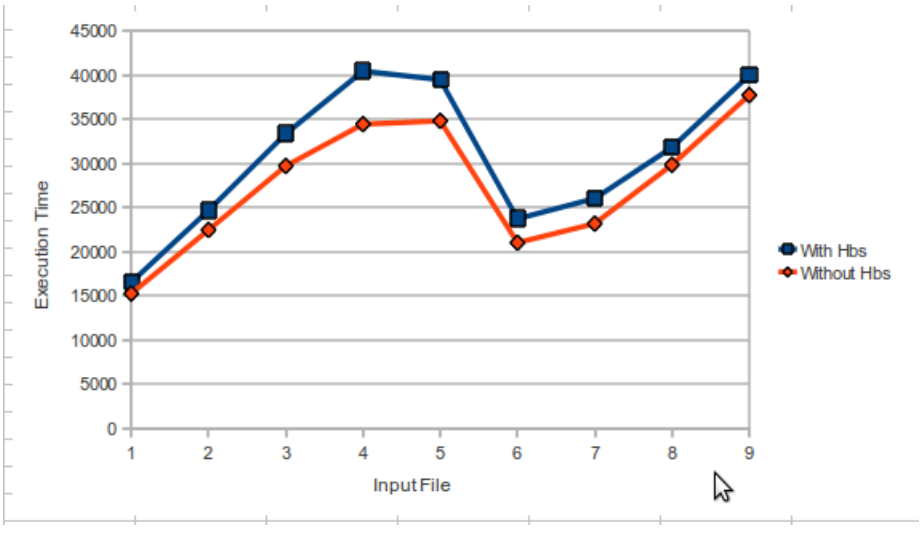

Figure 5.2: Overhead on application execution time - MiniMP3 decoder

Figure 5.1 and Figurer 5.2 describes the overhead on application execution time using MPG123 decoder and MiniMP3 decoder respectively.The results pointed out that the overhead of Application Heartbeats is relatively high. One of the reason leads to this effect could be the Application Heartbeats framework communication model base on file structure we are using for our project. Read/write file operations are high cost operations in the context of mobile device. Moreover, the number of heartbeats issued depends on the number iteration, which then depends on the size of data we retrieved in each iteration. It is obviously that the smaller the size of data in retrieved in each iteration, the more iterations will be executed, thus the more heartbeats will be issued. Therefore, more read/write operations will be used which cause the increase of impact. We can alter the size of retrieved data in each iteration to reduce the number of iterations, thus reduce the number of heartbeats but that is not the desired solution because Application Heartbeats framework should not require any change in application. However, in the particular situation like our MP3 decoder with playback enabled the situation can be brighter. In fact, playback is a non blocking operation, which means after finished writing data to the playback device's buffer, MP3 decoder is free to decode another frame. We can use the interval for playback to use up its buffer to decode another block and issue heartbeats without causing any delay to the application progress. Indeed, experimental test result 5.2 showed that with playback enabled, the impact of Application Heartbeats framework on Mp3 decoder application execution time is almost zero.

Table 5.2: Application Heartbeats overhead - playback enabled

	MiniMP3		
	With HBs(us)	W/o HBs(us)	Impact(%)
2.1Mb	27420	27420	0.0
3.3Mb	21500	21223	1.3
4.3Mb	28210	28114	0.3
5.3Mb	34720	34522	0.6
6.4Mb	33740	33560	0.5

The Figure 5.3 verify that the MPG123 algorithm has a greater performance than MiniMP3 algorithm on the same input file. It reflected the truth because MPG123 algorithm uses assembly code to speed up the decode while MiniMP3 does not.

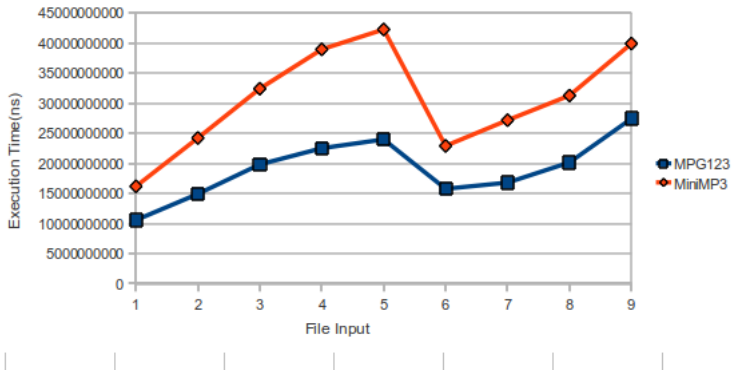

Figure 5.3: Performance comparision between two algorithms

5.2 Test Self-Adaptivity

5.2.1 Test purpose and expected result

This test aims at verifing the benefit of self-adaptive software system in comparision with non-adaptive software system. We built two scenarios for testing:

- Self-Optimization scenario : In this scenario, we are going to simulate the low-bandwidth condition by put some delay interval to be in direct ratio to the size of retrieved data in each iteration. The tests will be executed with and without self-adaptivity being enabled. In case non self-adaptive application, the music will be played interruptedly, because application need to wait for enough data before it can playback. We expect that with self-adaptivity being enabled, MP3 decoder will recognize the falling of heart rate and self-optimizes the size of data retrieved in next iterations to a desirable value, thus we are able to prevent application from being blocked for fixed intervals waiting for enough data.

- Optimization by external observer scenario : In this scenario, our application does not aware of the context but let a Consensus Object monitors and makes decision. We also use the same trick to simulate the falling in heart rate. We are going to add some delay to simulate the performance downgrade of MiniMP3 decoder (e.g., FPU stops functioning) during a specific interval. The Consensus Object monitors is expected to recognize the decreases in application's perfor-

mance, and issues appropriate command to application's service. The command must trigger the service to switch underlying algorithm implementation so that target application's heart rate can be recovered to desired value.

5.2.2 Testing conditions

The experiment was undertaken in generic condition:

- Software condition: A generic build of Android System image was built for testing purpose.
- Hardware condition: Instead of using real device, our tests was undertaken on emulator provided along with the SDK. The main hardware configuration options are :
  - Processor: ARMv7 CPU.
  - Cache's size: 66Mb.
  - RAM's size: 96Mb.
  - Max Virtual Machine application heap size: 16Mb.
  - SD Card : No.
  - Internal Storage : 32Mb.
  - Audio Plackback support : Yes.
  - Cache partition: yes.

5.2.3 Experimental Result and Comments

The first scenario test was undertaken using MiniMP3 decoder with 0.5 second delayed for receiving each decoded block. With self-adaptivity being disabled, application has shown a continous interruptions in playback. Whereas, the Figure 5.4 depicts application's heart rate with self-adaptivity being enabled. As can be observed, after a few adjustments, application's heart rate fell back into the interval between expected the minimum and maximum value (e.g., minum value is 20, maximum value is 30). This reflected the truth that user perceived some interruptions while application was performing adaptation then playback music went smoothly again. Thus, the result has shown the potential benefit of self-adaptive software system in case of self-optimization.

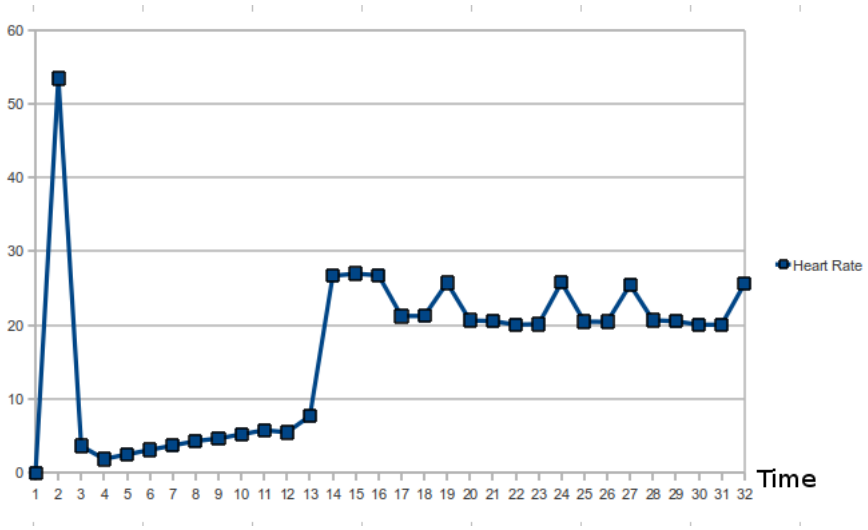

Figure 5.4: Self-optimization MP3 decoder

The Figure 5.5 depicts the application's performance change in second scenario. As can be observed from the graph, the dark blue line represents

application heart rate without external observer being enabled while the red line represents the application heart rate with external observer being enabled. The dark blue line showed that application had to endured performance degrade represented by a wide sunken part of the grapth. Whereas, the red line explained a short degrade in application performance (e.g., the graph went down for a very short interval). From user's point of view, they perceived some interruptions in both case but with self-adaptivity enabled application, music went smoothly again while it could not be so in the other case. It is obviously that external observer had expressed its ability in monitoring application and in making decision to make target application adapt to the new context.

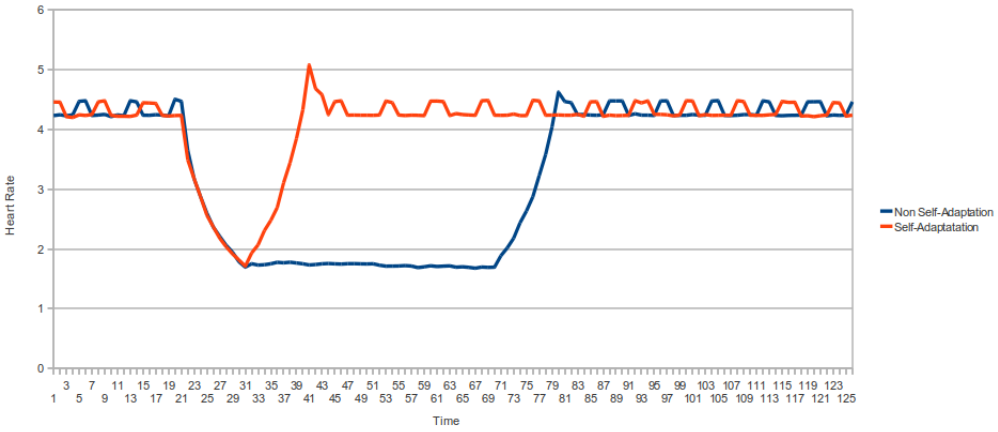

Figure 5.5: Optimization using external observer

5.3 Summarization

In this Chapter, we have already represented to readers the testing result of the project. Analyzing testing data of using Application Heartbeats

framework, we concludes that the framework adds notable overhead on application's execution time. There are two reasons leads to this overhead problem: the used file model by framework and Application Heartbeats framework is sensitive to the number of iterations executed by the critical loop. The result also proved that self-adaptivity provided potential benefit in both self-optimization scenario and optimization with external adaption engine scenario. Therefore, we have a strong base to believe that self-adaptive software system is the right answer to deal with skyrocketing complexity problem in modern systems.

Chapter 6

Conclusions and Future work

This thesis aims at studying one of the solution for solving the problem of skyrocketing complexity in modern computing system, the self-adapting software systems. Although this is just a partial solution with the scope limited to software solutions, its potential benefit has been proved in building autonomic system and reducing human efforts needed due to (re-)configuring and maintaining the systems.

In the scope of this dissertation, we first introduced the problem in Chapter 1; provided the overview about self-adaptive autonomic system as the general answer to skyrocketing complexity problem. Then we provided more details into the Software Engineering area, introducing methods for eliciting requirements while design a self-adaptive software systems as well as providing a taxonomy based on various aspect as in Chapter 2. The Chapter 3 introduced to readers a deeper view into self-adaptive software system architecture by representing the the control loops and related technique for building each process of the loop. Among those tech-

niques, we concentrated on introducing the Application Heartbeats Framework because it provides a very flexible, portable, simple way for application to define its goal as well as monitor its progress towards goal. Taking Application Heartbeats framework as a starting point, we demonstrated the logic of other process, and introduced the need and the implementation of the Consensus Object as a center element. The Consensus Object is the central entity that gathers all the information coming from Heartbeats-enabled applications and decides which actions to undertake in order to make the processes reach the desired goals. An extension definion of actuators called Service was also represented to readers. Service decoupled the application and its adaptable part, thus provided a more flexible in implementation. The Consensus Object controls Services using Service API, enables the adaptivity property of application.

After the dicussion in Chapter 3, we represented to readers the implementation of a self-adaptive MP3 decoder running on Android mobile operating system as an illustrative example. As argued in Chapter 3, there are amount of elements that can affect performance on a system. Among those elements, application knobs and implementations are likely to be the most common effects. Taking in to account of this consideration, in Chapter 4, we built and test an example of Implementation Library to certify dicussed concepts in Chapter 3.

The application then was put to test and the result was represented in Chapter 5 shows the potential benefit of self-adaptive software system in comparision to the original one. Testing was also taken to see the impact

and cost while using Application Heartbeats Framework as monitoring infrastructure. It is obviously that injecting Application Heartbeats into application, we have to endure some overhead but the result showed that overhead are acceptable in comparision to the benefit it can produce.

Successfully in one simple project on Android operating system provided us a strong base to go further in implementing more sophisticated applications as well as what we call self-adaptive support components. The following ideas can be accomplished in the near future:

- Provide self-adaptivity for Android's core components.
- Automatic the development process of self-adaptive software system. By saying automatic the development, we mean the development of libraries which have already intergrated ODA control loop. Developers can either fully rely or partial rely on those libraries while developing their own self-adaptive software system.
- While the idea above is likely take alot of efforts, we can provide to developer at Android Application layer the access to Application Heartbeats Framework which is located at Android Libraries layer through using JNI calls. It is the mediate way for developing self-adaptive software systems.
- It is also a great idea to take researchs to improve the performance of decision engine. So far, we still rely on human knowledge on choosing the best solution among available solution. An atonomic and efficient mechanism should be proposed to accomplish this task.

Bibliography

[1] Moore Gordon E. Cramming more components onto integrated circuits. *Electronics Magazine*, pages 4–4, 1965.

[2] Mazeiar Salehie and Ladan Tahvildari. Self-adaptive software: Landscape and research challenges. *ACM Transactions on Autonomous and Adaptive Systems*, 4(2):1–42, 2009.

[3] R. Laddaga. *Active software. In Proceedings of the International Workshop on Self-Adaptive Software*. 2000.

[4] R. Laddaga. Guest editorâs introduction: Creating robust software through self-adaptation. *IEEE*, 14(3):26–29, 1999.

[5] J.O. Kephart and D.M. Chess. The vision of autonomic computing. *IEEE*, 36(1):41–50, 2003.

[6] P.K. McKinley and M. SADJADI. Composing adaptive software. *IEEE*, pages 56–64, 2004.

[7] D.C. Schmidt. Middleware for real-time and embedded systems. *Comm. ACM*, 45(6):43–48, 2002.

[8] IBM-AC. Autonomic computing 8 elements. *IBM*, pages –, 2001.

[9] TRIVERIO MARCO. Application heartbeats: a technique for enhancing system self-adaptability. 2010.

[10] P. Oreizy, N. Medvidovic, and R.N. Taylor. Architecture-based runtime software evolution. In *In Proceedings of the International Conference on Software Engineering.*, pages 178–186, 1998.

[11] M.M. Kokar, K. Baclawski, and Y.A. Eracar. Control theory-based foundations of self-controlling software. *IEEE*, 14(3):37–45, 1999.

[12] R. Laddaga. Self adaptive software problems and projects. In *In Proceedings of the IEEE Workshop on Software Evolvability.*, pages 3–10, 2006.

[13] M. Jelasity, O. Babaoglu, and R. Laddaga. Interdisciplinary research: Roles for self-organization. *IEEE*, 21(2):50–58, 2006.

[14] D.M. Serguendo. Self-organisation: Paradigms and app. In *In Proceedings of the Engineering Self-Organising Applications Workshop.*, pages 1–19, 2003.

[15] R. Laddaga. *Self-adaptive software.* Tech. Rep. DARPA BAA, 98-12 edition, 1997.

[16] K.J. Lieberherr and J. Palsberg. *Self-adaptive software.* Engineering adaptive software, projest proposal edition, 1993.

[17] J.O Kephart and W. Walsh. An artificial intelligence perspective on autonomic computing policies. In *In Proceedings of the IEEE International*

workshop on Policies for Distributed Systems and Networks., pages 3–13, 2004.

[18] H. Liu, M. Parashar, and S. Harisi. A component-based programming model for autonomic applications. In *In Proceedings of the International Conference on Autonomic Computing.*, pages 10–17, 2004.

[19] J.P. Loyall, D.E. Bakken, R.E. Schantz, J.A. Zinky, D.A. Karr, R. Vanegas, and K.R. Anderson. Qos aspect languages and their runtime integration. In *In Proceedings of the International Workshop on Languages, Compilers, and Run-Time for Scalable Computers.*, pages 303–318, 1998.

[20] R. Sterritt. Autonomic computing: the natural fusion of soft computing and hard computing. In *In Proceedings of the IEEE International Conference on Systems, Management and Cybernetics.*, pages 4754–4759, 2003.

[21] D. Garlan and B. Schmerl. Model-based adaptation for self-healing systems. In *In Proceed-ings of the Workshop on Self-Healing Systems*, pages 27–32, 2002.

[22] G. Karsai and J. Sztipanovits. A model-based approach to self-adaptive software. *IEEE*, 14(3):46–53, 1999.

[23] S. Dobson, S. Denazis, A. Fernandez, D. Gati, E. Gelenbe, F. Massacci, P. Nixon, F. Saffere, N. Schmidt, and F. Zambonelli. A survey of autonomic communications. *ACM Trans.*, 1(2):223–259, 2006.

[24] Hoffmann Henry, Eastep Jonathan, Santambrogio Marco, Miller Jason, and Agarwal Anant. Application heartbeats for software performance and health. pages 1–12, 2009.