CHAPTER 3

ENHANCED SELF-ADAPTIVE DYNAMIC SOFTWARE ARCHITECTURE

3.1 INTRODUCTION
Software systems are growing complex due to the technological innovations and integration of businesses. There is ever increasing need for changes in the software systems. However, incorporating changes is time consuming and costly. Self-adaptation is therefore being the desirable feature of any software that can have ability to adapt to changes without the need for manual reengineering and software update. To state it differently robust, “self-adaptive dynamic software architecture” is the need of the hour. Unfortunately, the existing solutions available for self-adaptation need human intervention and have limitations. The architecture like Rainbow achieved self-adaptation. However, it needs to be improved in terms of quality of service analysis and mining knowledge and reusing it for making well informed decisions in choosing adaptation strategies. In this chapter Enhanced Self-Adaptive Dynamic Software Architecture (ESADSA) is proposed and implemented which provides automatic self-adaptation based on the runtime requirements of the system. It decouples self-adaptation from target system with loosely coupled approach while preserves cohesion of the target system. We built a prototype application that runs in distributed environment for proof of concept. The empirical results reveal significance leap forward in improving dynamic self-adaptive software architecture.

3.2 BACKGROUND
Software systems drive the business in this age of digital world. The modern software systems should be equipped with highly desired features in a distributed environment. Therefore, software systems must become more “versatile, flexible, resilient, dependable, service-oriented, mash-able, inter-operable, continuously available, robust, decentralized, energy-efficient, recoverable, customizable, configurable, self-healing, configurable and self-optimizing” by adapting to changing operational contexts and environments.
Traditional software is implemented under static decisions in analysis and design time based on assumptions about the requirements and runtime environment. Therefore, any unexpected changes to the requirements or runtime environment will lead to a manual maintenance process, which is intolerable in critical systems. The existing self-adaptive software architectures are utility based and making them quality-aware is a challenging problem to be addressed. “Self-adaptive software” modifies its own behavior at runtime in reply to changes in its “operating environment”. By “operating environment”, we mean anything noticeable by the software system, such as end-user input, external hardware devices and sensors, or program instrumentation. Application developers must answer numerous questions when developing a “self-adaptive software system”. Under what conditions does the system undergo “adaptation”, Should the system be “open-adaptive” or “closed-adaptive”, What type of autonomy must be supported, how often is “adaptation” considered, under what circumstances is adaptation cost-effective.

A system might, for example, modify itself to improve system response time, recover from a subsystem failure, or incorporate additional behavior during runtime. A system is “open-adaptive” if new application behaviors and adaptation plans can be introduced during runtime. A system is “closed-adaptive” if it is self-contained and not able to support the addition of new performances. An extensive range of autonomy might be needed, from fully automatic, self-contained adaptation to human-in-the-loop. An extensive range of strategies can be used, from opportunistic, continuous adaptation to lazy, as-needed adaptation. The benefits gained from a change must compensate the costs associated with making the change. Costs include the performance and memory overhead of monitoring system behavior, determining if a change would improve the system, and paying the associated costs of updating the system configuration. An extensive range of policies can be used, from unceasing, accurate, current comments to sampled, rough, historic observations.

The approaches found in the literature have certain limitations and issues that need to be resolved. Many were addressed by Rainbow framework [51] such as exception handling and balance between local and global perspectives. It also focuses on quantity of adaption with many building blocks to achieve self-adaption. Customizable elements and reusable
infrastructures are the two good features of Rainbow. However, the Rainbow framework has certain limitations as described here. Utility based theory was used for best “adaptation” path under ambiguity. The utility-based frameworks are not fully quality-aware. “Quality of Service” (QoS) analyzers can be built to uninterruptedly monitor for improvement chances. The historical information usage is not sophisticated in the existing frameworks. It can be improved with state-of-the-art data mining techniques for improving decision making. All these issues are overcome in this chapter with required modules incorporated into the framework.

3.3 SELF ADAPTIVE SOFTWARE

“Self-adaptive software” adjusts its own performance in reply to changes in its “operating environment”. By “operating environment”, we mean anything noticeable by the software system, such as “end-user” input, external hardware devices and sensors, or program instrumentation. A software system has a set of properties, including its internal properties, its input and output, and the relationships between the inputs and outputs. These properties are captured in the form of specifications, as the desired properties of the software system. However, in reality the operational software may not fully match its specification. Moreover, as a software system ages, its environment and user needs will change, which will result in an increase in the deviation of the software’s properties from its desired properties? This phenomenon is captured by the first law of software evolution by Lehman [1]: “Software change is inevitable and systems must be continuously adapted.” In classic software engineering, all required post-delivery changes are handled through evolution changes that result in a new software system. In the case of “self-adaptive software”, some of the required changes can be handled through adaptation changes, which can change the properties of the software dynamically without generating a new software system.

The adaptation changes impose less cost compared to their equivalent evolution changes, and are usually invertible. In case the original software system is not adaptive, we can apply a set of evolution changes, which serve as adaptive traits that increase the adaptedness of the software system. As the environment and/or user needs change, the desired properties of the software including its adaptedness may change overtime. Hence, evolution changes
that introduce and/or improve adaptedness are not one-time changes: We need continuous evolution to maintain the desired adaptedness in software. Each change has a cost. The effectiveness of a change is determined by the degree to which the change encounters its purpose of achieving the desired properties of the software throughout the software’s lifespan.

The objective of systems engineering is to ensure that a system is designed, built, and operated so that it achieves its purpose in a “cost-effective” manner. “Cost-effectiveness” considers both the cost and the effectiveness of a system in the context of the system objectives. If a SAS system has been engineered and used properly, it can greatly improve the 4 cost-effectiveness of software change through its lifespan. However, in practice, many of the existing approaches towards SAS are rather expensive and may increase overall system complexity, as well as following future maintenance costs. This means that in many cases SAS is not a good solution, because its development and maintenance costs are not being paid off. The situation is even worse in the case of making current (legacy) systems adaptive. Hence, the most important question to be answered is: “How can we effectively engineer and employ self-adaptive software?” There are several factors that have an impact

**Figure 3.1:** Self-adaptive reference software architecture [9]
on the cost-effectiveness and usability of self-adaptive software; however, the main objective of this thesis is to make a software system adaptive in a cost-effective way, while keeping the target adaptive software generic, usable, and evolvable, so as to support future changes.

3.4 NEED FOR SELF-ADAPTIVE SOFTWARE

As software ages, it should change with time, or else it will not survive [2]. Software changes can be grouped as follows, based on the purpose that they serve: They may be [2] remedial for fixing bugs, adaptive for adapting the software to new environments, perfective for updating the software according to requirements changes, and preventive for making the software more maintainable [3].However, software maintenance and evolution are inevitably costly and time-consuming activities for almost any software-intensive system. From an economic point of view, system adaptation, on average, soaks up most of the post-delivery non-corrective software costs (~ 80% of total post-delivery costs are non-corrective) in the life of a software system [4]. Moreover, a key characteristic of the maintenance/evolution of large software systems is that change becomes increasingly difficult and expensive over time [5]. Since the era of software engineering started, many technologies and development paradigms have been introduced to scope, estimate, control, and improve the predictability and efficiency of software change [6], [7], [8]. These approaches aim at reducing the cost of change by either (i) managing the total number of the required changes, (ii) making software change easier (e.g., creating more flexible and variable software), (iii) reducing software down-times, and (iv) predicting the change. A promising solution to reduce the cost of software change is to develop self-adaptive software (SAS) systems that are able to manage changes dynamically at runtime in a rapid and reliable way. One of the main advantages of SAS is its ability to manage the complexity that stems from highly dynamic and nondeterministic operating environments. Frequently changing user needs or high variability in the amount of available resources are two possible scenarios in which SAS can reconfigure and continuously optimize itself at runtime. An adaptive behavior can replace a normal behavior in an autonomous way. In general, a SAS follows a reference architecture that comprises two main subsystems [9]: (i) the “adaptation manager (autonomic manager)”, and (ii) the “adaptable software (managed element)”. The
“adaptation manager” acts as an external controller that observes the adaptable software for changes in its operating states and selects appropriate adaptive behavior accordingly. These two subsystems are connected through sensor and effectors interfaces.

3.5 PROPOSED “SELF-ADAPTIVE DYNAMIC SOFTWARE ARCHITECTURE”

In this section we describe the proposed “self-adaptive dynamic software architecture”. Self-adaptive software architecture can cater to the dynamic needs of the software at runtime. It can adapt to runtime situations. Select adaptation needs to work for different kinds of systems and quality requirements. The adaptation is to be made with explicit operations that are chosen at run time. Such architecture should provide an integrated solution that saves time and effort of engineers as it can adapt to situations without the need for writing code and update the software explicitly. The framework satisfies these requirements by supporting many mechanisms that lead to dynamic self-adaptation. Our architecture is known as Enhanced Self Adaptive Dynamic Software Architecture (ESADSA). This work has been influenced by Rainbow framework [31]. Our architecture shown in Figure 1 extends rainbow framework with two additional modules. They are known as Quality of Service (QoS) analyzers and History/Knowledge Miner.

![Figure 3.2: The proposed framework named ESADSA](image)

53
The framework has two layers known as architecture layer and system layer. Broadly, architecture layer represents self-adaptive mechanism which is made up of many components that work in tandem with each other. The system layer represents the target system and the plumbing components that are used to realize self-adaptation. There are two mechanisms for monitoring the target system. They are known as probes and gauges. The remarks are reported to model manager. The architecture evaluator component is responsible to evaluate the model when model gets updates. It checks architectural constraints within acceptable range. When evaluation concludes that there is a problem in the system, the evaluation management invokes adaptation manager. The adaptation manager is responsible to initiate “adaptation process” and select a suitable “adaptation strategy”. Then the strategy executor comes into picture in order to execute the strategy on the runtime system. This is achieved through system level effectors that ensure realization of changes to the target system through self-adaptation.

There are three important aspects that are used to realize the software architecture. They are known as software architecture, control theory, and utility theory. Self-adaptation is possible with proposed software architecture that makes it cost-effective. Control theory and mechanisms ensure smooth adaptation. Utility theory helps in finding best strategy in self adaptation. Moreover, the proposed architecture has two components for optimization. These components are known as QoS analyzers and knowledge miner.

3.5.1 Translation and Monitoring

The translation infrastructure provided in architecture takes care of monitoring and action. This will help in connecting the gap between the target system and architecture layer. It has monitoring mechanisms like “probes and gauges”. These mechanisms get system states and update the model from time to time. The probe is responsible to measure target system while the gauge is responsible to interpret the measure identified. The probing is done on the attributes such as process run time and CPU load.

The solution given in [51] has certain limitations. They are described here. Utility based theory was used for “best adaptation” path under uncertainty. The utility-based frameworks
are not fully quality-aware. “Quality of Service” (QoS) analyzers can be built to continuously monitor for improvement chances. The historical information usage is not sophisticated in the existing frameworks. It can be improved with state-of-the-art data mining techniques for improving decision making. The modules incorporated in the framework for overcoming the drawbacks of [51] are as follows. These modules improve the performance of the architecture in making it robust and dynamic and self-adaptive software architecture.

3.5.2 QoS Analyzers
- These are the components in the architectural layer of the proposed framework.
- They take care of quality-aware analysis to exploit opportunities at runtime based on the QoS parameters like resource utilization, response time, etc.

3.5.3 Building & Mining Knowledge Base
A holistic historical information maintenance and analysis is made using state-of-the-art data mining techniques for making well informed decisions towards best adaptation path.

3.6 NEWS.COM WITH SELF ADAPTATION
To demonstrate the usefulness of the proposed framework a hypothetical news web site by name News.com is taken as case study. This application is distributed in nature. It is n-tier application that has web-based client. The tiers in it include “client tier”, “web tier”, “business tier” and “data tier”. It has a load balancer which will balance the load across the servers. In other words, the service is given from different servers. There are web-based clients to the application where the clients make stateless requests without a session-based approach. The requests are processed by different servers based on their availability.
As can be seen in Figure 3.3, the browser is in the client tier using which users can make request for news. Web server is in web tier which invokes servlet that is responsible to process the request. However, the servlet cannot render news directly. It invokes a remote server program known as RMI server. The RMI server is in business tier where business rules are applied. The RMI server interacts with database which is the sources of the news. The tomcat server is a web server which has the following service architecture.
As can be seen in Figure 3.4, the tomcat server has support for different services and connectors. The connector may be coyote non-ssl or warp on 8080 port number. The services may be tomcat-standalone or tomcat-apache. Each service can make use of an engine that has required mechanism to have virtual hosts running different web applications. Web application is executed in tomcat server and servlet are executed by the servlet container or web container located in tomcat server.

The business model of “News.com” is to provide information about news to its customers with good response time and with optimal cost at the “server pool”, with operational budgets. However, the objectives of the News.com are not fulfilled when there are sudden spikes in the number of news requests from its clients due to certain popular events. This has resulted in unacceptable latencies and reduced customer satisfaction. The administrators of the application need to do certain configuration settings manually. They can either increase the size of server pool or shift to different content mode. When there is heavy load the administrators make decisions and switch to text mode keeping cost and budgets of server pool in mind. The adaptation is done manually with the following objectives.

➢ The default content mode of the servers is multimedia
➢ When situation demands switch to textual model
➢ Increase the server pool size as and when required
➢ Decrease the server pool size as and when required

Performing these things manually is time taking and error prone. In this chapter, automation of this process is made by implementing the proposed framework to make the News.com application “self-adaptive”. Making well informed decisions automatically with tradeoffs among different objectives is a challenging task handled by the proposed framework.

In case of News.com the server response time is monitored and measured in the architecture model. This is done by model manager. The architecture evaluator ensures that the latency is not above given threshold. When it is not meeting the condition, the evaluator triggers the adaptation manager to make necessary steps in order to activate more servers or switch to text mode to reduce content quality. The system hooks are used by the strategy manager to make use of strategy that can handle the situation.
The proposed framework is implemented using Java platform to realize self-adaptive dynamic architecture for News.com. The implementation of the framework is to ensure the “self-adaptation”. It is independent of any application like News.com. It can be employed to any application that runs in the environment.

As shown in Figure 3.5, there are many clients making simultaneous requests to web server. The adaptation framework is there to monitor the requests and ensure that the servers are automatically configured. There are four important operations done here. They include increasing server pool size, decreasing server pool size, switching to text mode and switching to multimedia mode. The “QoS analyzer” proposed in the framework takes care of quality analysis. This knowledge coupled with the history miner’s outcome can bring about best possible solution for “self-adaptation”.

3.7 EXPERIMENTAL RESULTS

Experiments are made with live News.com application built in the laboratory. It is executed in distributed environment using different machines connected to LAN. The number of requests is simulated and the latency and other observations are recorded.
As shown in Figure 3.6, the trends in adaptation time when number of requests are increased or decreased is presented. The “horizontal axis” shows number of requests used in different experiments, while the “vertical axis” shows the adaptation performance.

Figure 3.7: Adaptation accuracy
The proposed system is evaluated with 100 experiments and found that it shows very negligible false positives. However, it is the proof that the system can be improved further in order to achieve 100% true positives.

**Figure 3.8:** Latency of random requests

The latency of different requests randomly chosen are observed and plotted in the graph shown in Figure 3.8. The latency time is presented in vertical axis while the number of random requests is shown in horizontal axis.

**Figure 3.9:** Quality of service in terms of throughput
As shown in Figure 3.9, the quality of service is measured in terms of throughput. The response time is considered with different number of requests made to the News.com servers. The latency is recorded and presented.

3.8 SUMMARY
Software systems have been evolving. The contemporary applications are able to drive businesses of multiple organizations that form as an integrated set of business. Applications in such environment need changes dynamically. The changes are to be incorporated in traditional approach by software engineers. However, this approach is costly and time consuming. To overcome this problem, many self-adaptive frameworks came into existence. Rainbow is one such architecture which enables dynamic self-adaptive software. However, the Rainbow architecture has specific limitations in terms of QoS analysis and knowledge mining in making decisions for adaptation. A self-adaptive dynamic software architecture named ESADSA is proposed and implemented which is based on a holistic approach with focus on QoS and knowledge mining for making expert decisions in adapting strategies as part of self-adaptation. We built a prototype application to demonstrate the proof of concept. Our results revealed that the proposed architecture significantly improves the self-adaptation performance. In chapter 4, the proposed self-adaptive architecture is elaborated with algorithms and evaluation using a real time case study.