AN AUTONOMIC SOFTWARE ARCHITECTURE

FOR DISTRIBUTED APPLICATIONS

by

Mohammad Muztaba Fuad

A dissertation submitted in partial fulfillment
of the requirements for the degree

of

Doctor of Philosophy

in

Computer Science

MONTANA STATE UNIVERSITY
Bozeman, Montana

June 2007

APPROVAL

of a dissertation submitted by

Mohammad Muztaba Fuad

This dissertation has been read by each member of the dissertation committee and has been found to be satisfactory regarding content, English usage, format, citations, bibliographic style, and consistency, and is ready for submission to the Division of Graduate Education.

Professor Michael J. Oudshoorn

Approved for the Department of Computer Science

Professor Michael J. Oudshoorn

Approved for the Division of Graduate Education

Dr. Carl Fox, Vice Provost

STATEMENT OF PERMISSION TO USE

Mohammad Muztaba Fuad

June 2007

To

My Mom

ACKNOWLEDGMENTS

I would like to thank my wife Debzani for all her support during the course of my Ph.D. She deserves more than a mere acknowledgement.

My advisor Prof. Michael J. Oudshoorn was always there for me. His support and encouragement rescued me from frustration and guided me to my research. Without his kind help it would have been impossible to finish this work.

I would like to express my gratitude to my Ph.D. committee members for their constant support. I am fortunate to do my Ph.D. in a department where all the faculty members showed interest in my work and gave constant feedback. I thank them for their active interest in my research.

I specially like to thank Ms. Jeannette Radcliffe, Ms. Kathy Hollenback and Scott Dowdle of Computer Science department for providing me with a wonderful logistical support for my research and education.

I like to thank Benjamin Ph.D. Fellowship foundation for awarding me with a fellowship. I thank Montana NASA EPSCoR for providing financial support for my research.

I would like to thank the people of Bozeman for providing me with a perfect atmosphere to finish my education. I am fortunate to be in the Big Sky Country.

Finally, I would like to thank my family and friends for always looking after me and wishing the best for me. Thank you.

TABLE OF CONTENTS

1. INTRODUCTION 1

Problem Specification 4
Rationale for Autonomic Systems 6
Motivation 8
Goals 10
Scope of This Research 12
Challenges 14
Methodology 16
Contributions 17
Outline of the Thesis 18

2. BACKGROUND 20

What is Autonomic Computing? 20
Why Java? 25
Which JVM? 29
Source Code or Byte Code? 30
Summary 30

3. JVM AND JAVA CLASS FILE 32

Structure of the JVM 32
 Data Types and Values 35
 Variables 35
 Fields and Methods 36
Structure of Java Class File 36
 Code Attribute 40
 Local Variable Attribute. 40
 Line Number Attribute 40
 Exception. 40
Byte Code Instructions 41
An Example 43
Byte Code Modification Tools 46
Summary 50

4. RELATED RESEARCH 51

Existing Middleware Technologies 51
Automatic Partitioning 54
Automatic Distribution 57
Autonomic Systems 58
Autonomizing Existing Systems 60
Autonomic Programming Environment 63
Autonomic Element Architecture 65
Self-healing Transformation 67

TABLE OF CONTENTS - CONTINUED

Other Related Autonomic Systems 68
How This Work is Different? 69

5. SYSTEM ARCHITECTURE 72

Requirements 72
Service Architecture 75
System Operation 78
Policies 80
Management Operations 82
Autonomic Properties 83
    Self-configuration. 84
    Self-healing. 86
    Self-optimization 87
    Self-protection 89
Distribution Transformation 89
Autonomic Transformation 92
Performance Analysis 98
Summary 104

6. AUTONOMIC ELEMENT ARCHITECTURE 105

Introduction 105
Architecture 108
Life Cycle 111
Implementation Details 114
    Data Structures 114
        Data Atoms 114
        Priority Queue 115
        Locker 116
        Data Streams 117
        Hooks 117
        Atom Repository 118
    Operation of the Sub-components 118
        Sensors and Effectors 119
        Sub-components 119
Performance Analysis 120
Summary 123

7. ADDING SELF-HEALING 125

Introduction 125
Self-healing Architecture 128
Faults and Fault Model 129
Code Transformations 133
Local Variables 137

TABLE OF CONTENTS - CONTINUED

Adding Checkpoints 141
Status Data 142
Task of Fault Analyzer and Fault Healer 144
Runtime Exceptions 145
Performance Analysis 147
Summary 150

8. CONCLUSIONS AND FUTURE WORK 152

REFERENCES 155

APPENDICES 164

APPENDIX A: Flow Charts of Different Processes 165

APPENDIX B: Test Program Listing 172

LIST OF FIGURES

Figure Page

1. Structure of an Autonomic Element........ 23
2. Structure of the Java Virtual Machine........ 33
3. JVM Runtime Data Area........ 34
4. Structure of a Java Class File........ 37
5. An Example Java Class........ 44
6. Constant Pool of the Given Class........ 45
7. Byte Code Instructions Inside the Methods........ 46
8. Autonomic Service Architecture........ 75
9. Hierarchical Management of System........ 77
10. Flow of Operation........ 80
11. System Startup Process........ 85
12. Object Transformation........ 95
13. Example Class Diagram........ 99
14. Transformed Runtime Program Structure........ 101
15. Sequence Diagram of Delegated Calls…….. 102
16. Timing Information for Delegated Invocation........ 103
17. Internal Architecture of the Autonomic Element........ 108
18. Autonomic Element Extensions........ 111
19. Autonomic Element Life Cycle........ 112

LIST OF FIGURES -CONTINUED

Figure Page

20. Structure of a Data Molecule.................. 115

21. Runtime Throughput.................. 121

22. Flow of Atoms.................. 122

23. Self-healing Architecture.................. 128

24. Representation of Fault Model.................. 132

25. Self-healing Transformation Example.................. 136

26. Local Variable Access Indexing.................. 138

27. Code Fragment to generate a New Local Variable Information.................. 139

28. Code Inflation Due to the Addition of Self-healing Primitives.................. 149

29. Execution Times of the Three Test Programs.................. 150

LIST OF TABLES

Table Page

1. Variables inside JVM.......... 36

2. Constant Pool Entries.......... 38

3. Java Class Files Attributes.......... 39

4. Academic Research Projects on Autonomic Computing.......... 69

5. Industry Research Projects on Autonomic Computing.......... 69

6. Effects of Multiple Calls on Respnose Time.......... 103

7. Code Statistics.......... 123

8. Faults and Default Actions.......... 131

9. Evaluation Framework for the Technique.......... 148

10. Code Inflation Due to Local Variable Addition (in bytes).......... 149

ABSTRACT

Autonomic computing is a grand challenge in computing that aims to produce software that has the properties of self-configuration, self-healing, self-optimization and self-protection. Adding such autonomic properties into existing applications is immensely useful for redeploying them in an environment other than they were developed for. Such transformed applications can be redeployed in different dynamic environments without the user making changes to the application. However, creating such autonomic software entities is a significant challenge not only because of the amount of code transformation required but also for the additional programming needed for such conversion.

This thesis presents techniques for injecting autonomic primitives into existing user code by statically analyzing the code and partitioning it to manageable autonomic components. Experiments show that such code transformations are challenging, however they are worthwhile in order to provide transparent autonomic behavior. Software architecture to provide such autonomic computing support is presented and evaluated to determine its suitability for a fully fledged autonomic computing system. The presented architecture is a novel peer-to-peer distributed object-based management automation architecture. In this model, independent or communicating objects are treated as managed elements in the geographically distributed autonomic elements.

The presented organization offers significant advantages over a traditional client-server organization by permitting the incorporation of self-management properties into each of the distributed nodes and making each of the Autonomic Elements in the distributed environment identical in terms of managerial capacity. The unification of traditional client-server roles allows management functions to be distributed across different elements in the system, allowing autonomous behavior of the whole system. This thesis also presents a self regulating design of an autonomic element in a distributed object environment. Architectural choices have a profound effect on the capabilities of any autonomic system and affect many of the design decisions during its implementation. The goal of this architectural design is to provide an easy to program autonomic element which can be implemented in most domains with minor modifications. Profiling and experimentation with this design shows that it is lightweight and perform smoothly without causing extensive overhead.

CHAPTER 1

INTRODUCTION

Today’s information technology landscape is bristling with innovations and changes. New technologies are rapidly emerging and new versions of existing technologies continue to be released. To keep pace, programmers need to quickly adapt their existing applications to new technologies. The race to be at the cutting edge of technology makes complexity a major issue in all aspects of information technology as new technologies are being incorporated into existing systems and overall behavior of the system becomes unpredictable. Software developers have exploited the rapid upsurge in computational power in every possible way, producing ever more sophisticated software applications and environments, which results in enormous growth in the number and variety of systems and components. As systems become a complex mesh of technologies, software architects are less able to anticipate and design interactions among components, which further results in complexity not only in those systems, but also in the environments they operate within. Although there have been attempts to reduce such complexities within systems by introducing better software engineering practices, the complexity remains as more and more new technologies and systems are being incorporated together. Such an environment is a complex, heterogeneous tangle of hardware, middleware and software from multiple vendors that is becoming increasingly difficult to program, integrate, install, configure, tune, and maintain. This leads to the idea of autonomic computing [42, 58] where the complexity and the management of such systems is handled by the system itself.

For computationally intensive, large, parallel applications there is much to be gained from globally distributing the application across a wide variety of machines and obtaining an increase in throughput and a performance speed-up. The difficulty is that the programmer wishing to execute such an application may not have the physical resources themselves, nor may they have the necessary skills to effectively distribute the application manually. Programming such a distributed application across a heterogeneous network is a complex and tedious task and programmers need expert knowledge of the underlying system to undertake such distribution. In addition, programmers of such distributed applications must explicitly handle all the associated distribution management issues, such as fault tolerance [99], load balancing [99] and resource allocation [99] by themselves. For average programmers, this is a daunting task. The emergence of loosely-coupled, but highly reliable, clusters of networked machines has motivated people to build powerful applications (not necessarily client-server applications) using a wide range of resources. Object oriented technology is currently the most widely accepted programming methodology convenient for a standard programmer to design and implement applications for standalone systems. For a programmer, it becomes tedious when it is necessary to manage the distribution aspects of the program as well. RMI [94], CORBA [70] and some other middleware platforms try to reduce this tedium to an extent. However to use them, programmers need advanced knowledge of programming interfaces and paradigms. In a distributed programming environment, the configuration of the distribution setting may vary from time to time, but the program interactions may remain the same. A system which could redeploy the application code according to resource availability and some other specifications and goals is certainly desirable.

Programmers may elect to build a distributed application from scratch and ignore any existing code or system which had previously been developed but not intended to run in a distributed fashion. In reality, one can not abandon an existing parallel application or re-implement it for the distributed platform. It is beneficial if it is possible to transform and retrofit an existing application so that it runs in a distributed fashion, without the programmer having to deal with the programming or the management issues related to distribution. These issues interact in complex ways and even a semi-automated system is unlikely to be as effective as a human expert in terms of reaching a near optimal solution. However, the vast majority of programmers lack the ability to undertake the distribution themselves, and few have the ability to find the optimal solution. Worse still, the optimal distribution is specific to an application and the manner in which it is used. In fact, the optimal distribution may not be realized as resources may be unavailable and network congestion may occur. In order to deal with such difficulties, the system must be adaptive and dynamic in nature – an ideal application for an autonomic computing system.

This thesis discusses an autonomic distributed system that explores the "all care and no responsibility" principle of distribution [75] whereby the average programmer does not wish to take responsibility for the physical distribution and coordination of the application but is, however, concerned with the application throughput and total execution time. There are several potential benefits of automatic program distribution over manual distribution such as increased throughput and productivity, correctness of distribution, adaptive execution of application and concurrency exploitation. It is recognized that this approach may not be suited for all computation. However, in many cases, it is crucial. One such broad area of applications is inherently concurrent data intensive programs such as image and animation processing, fluid mechanics and universe exploration models. In the context of all these application domains, autonomic distributed systems promise to assist with an increase in accessibility, resource sharing, throughput and scalability and reduction in execution time and programmer involvement to manage such systems. The other facet of this problem is when the source code of such application is unavailable [83]. The question then arises of how to inject autonomicity to an existing non-autonomous system. Adding autonomic properties into existing applications is immensely useful for redeploying them in an environment other than they were developed for. Such transformed applications can be redeployed in different dynamic environments without the user making changes to the application. This would be of great help to the users of such system, if they do not have to worry about the consequent transformations and the management of corresponding runtime infrastructure.

Problem Specification

When building a distributed application from scratch, programmers can use their own framework and any programming language that they choose and could ignore existing applications. An existing application may be originally developed with the intent that it be executed on a single host and whose source code may no longer be available. In reality, one can not abandon an existing parallel application or re-implement it for the distributed platform from scratch because of the costs involved in implementing significant software systems. Although, there are different programming tools for distributed programming, these programming tools and environments are mainly for developing new distributed application from scratch. These programming tools do not adapt existing software to distributed execution on multiple hosts. They are not helpful in modifying the existing software so that it executes in a distributed fashion. The programmers have to manually modify the source code of such programs and follow specific programming conventions. This transformation is not only time consuming and error-prone, but also requires advanced programming knowledge. The problem is aggravated when the source code for such an application is no longer available for transformation. The challenge is then to perform all such transformations at the object level instead of the source code level.

It is not always a feasible option to build self-adaptive distributed systems from scratch. Mostly because of the cost and time associated with such a major development. Programming such a distributed application is also a tedious task and programmers need expert knowledge to handle the distribution management issues along with application specific domain knowledge for the problem at hand. For average programmers, this becomes a daunting task when they also have to incorporate autonomic primitives into the system. In general, programmers want to concentrate on the problem in hand, rather than spend time on incorporating autonomic behavior into their system. It is tremendously beneficial to programmers if such autonomic behaviors can be added automatically and transparently to existing systems. Although object oriented technology provides programmers with the advantage of rapid program development in a networked environment, it often becomes overwhelming when it is necessary to handle the autonomic computing aspects of the program as well.

Since program behavior is highly dynamic and a different distribution configuration may be appropriate in different phases of the execution of a program, components should be autonomic in behavior to increase productivity and system manageability. Such autonomic applications can be redeployed in different dynamic environments without the user making changes to the application. However, there are many challenging aspects [74] associated with such program transformations and incorporation of autonomous behavior into existing programs. Such a distributed system is envisioned as an Autonomic Computing [42] challenge where *Autonomic Elements* (AE) [42, 58] are incorporated automatically to handle the complexities associated with distribution, coordination and efficient execution of program components.

Rationale for Autonomic Systems

Although the phrase "autonomic computing" is new to the computing arena; the goal of designing and implementing self-managed systems has existed for a long time. A few examples include:

- Personal computers now have some self-configuration abilities to receive updates from the Internet and update the corresponding software or even the operating system.
- Redundant Array of Independent Disk (RAID) systems have some self-healing capabilities. One type of RAID systems (RAID Level 1) mirror data between two disks at the same time. If either disk fails, the other continues to function as a single disk until the failed disk is replaced.

While these technologies move in the right direction, they are just a precursor to complete autonomic computing. As more and more complex software applications and environments are developed and systems are distributed, it becomes increasingly difficult to maintain those systems or even develop such systems. In such a complex mesh of technologies, it is difficult to anticipate interactions among different systems. Although there have been attempts to reduce such complexities within systems by introducing better software engineering practices, the complexity remains as more and more new technologies and systems are incorporated together.

As computing systems continue to evolve to meet ever changing needs, dynamic computing systems which are self-manageable give business and scientific organizations the ability to automate their business and scientific tasks. As these systems are an interconnected mesh of different technologies, an autonomic approach is desirable as new application modules and functions can be added or removed at a faster rate without jeopardizing the overall functionality of the system. As computing infrastructure is now distributing processing power (as in Grid computing [38]) and increasingly more and more public networks are accessible for use in such environments, programmers need to consider rapid changes in the configuration and associated unpredictability along with system wide management issues. The autonomic computing approach is the only viable solution in such a diverse and uncertain environment.

If the business and the scientific communities fail to adapt autonomic computing or similar technologies in the immediate future, there are significant consequences:

1. Systems will be more complex than ever, and will be increasingly difficult to program, integrate, install, configure, tune, and maintain.
2. More and more highly skilled IT personal will be needed for such systems.
3. The consequent cost will be substantial and senior management will reject those astronomical budget requests.

4. Reliability and performance of system will deteriorate and the cycle of problems will continue.

Fortunately, much of the business and scientific community recognizes this looming crisis and agrees on self manageable systems, which operate in accordance with high level objectives set by humans, as the most promising solution. Although IBM envisioned autonomic computing in 2001 [42], Sun Micro System's *N1* [97], Hewlett-Packard's *Adaptive Enterprise Initiative* [41], Microsoft's *Dynamic Systems* initiative [65], Intel's *Proactive Computing* [46], Cisco's *ASF* [16], are all related industry efforts that recognize that self-manageable systems are vital for the future of computing. There are also significant research activities proceeding within the scientific community on this topic (Chapter 4 provides extensive examples). Although it is difficult to envision what the next era of computing will bring, success will be achieved when users do not need to think of the management of their computing systems.

Motivation

There have been substantial changes in computing practices in recent years mainly because of the proliferation of low priced powerful workstations connected by high-speed networks. However, it has been noted that many of these workstations are idle or lightly loaded most of the time [98]. There is a great potential if one could effectively harness the computation power of these under-utilized workstations. The most obvious motivation is that the resulting system can have much larger memory and processing capability than any standard computer. However, to secure the associated benefits, excessive programmer involvement is needed. For an average user, the task of distributing a large computation across a heterogeneous environment proves to be a tedious and cumbersome process. Therefore providing users with a self-adaptive distributed environment that automatically and transparently distribute and manage application across the available resources would be of great help.

Although it may be desirable to build such self-adaptive distributed systems from scratch, it is not always a feasible option, mostly because of the cost and time associated with such a major development, but also because it is not practical to abandon an existing distributed or parallel application and re-program it from scratch to be self-adaptive. Programming such a distributed application is also an error-prone task and programmers need expert knowledge to handle the distribution management issues along with programming the application problem at hand. For average programmers, this becomes a daunting task when they also have to incorporate autonomic primitives into the system. In real life, programmers want to concentrate on the problem in hand, rather than spend time on incorporating autonomic behaviors in their system. It is helpful to programmers if such autonomic behaviors can be added automatically and transparently to existing systems. Although object oriented technology provides programmers with the advantage of rapid program development in a networked environment, it becomes overwhelming when it is necessary to handle the autonomic computing aspects of the program as well.

Huge data and computationally intensive, collaborative e-science research applications, such as computational biology, bioinformatics and distributed data mining, are on the rise in recent years in both academic and industry research domains. Each of these types of applications demands a high-performance, dynamically reconfigurable distributed system and an easy to use middleware infrastructure to enable successful achievement of research goals. As scientific groups look to explore larger and more complex systems, their need for computing support increases. This research addresses the shortfalls of current distributed technologies by adding autonomic features to the system and by ensuring efficient utilization of resources across the network. This improved support and self manageability in the system meets the performance and dependability demands of these research groups along with making their goals more easily achievable.

The motivation for this research is to enable highly collaborative engagement of researchers in different domains by seamlessly utilizing the vast array of computing resources made available through the creation of self manageable cyber infrastructure. While centralization of certain large-scale research facilities is critical to achieve economies of scope and scale, methods for making these resources widely and easily available are equally important in meeting the goals of different scientific research groups.

Goals

This research aims to develop an autonomic computing architecture to distribute user applications across the available resources and provide autonomic infrastructure support to the applications programmer. A system is presented where the system analyzes the existing user code and suggests tasks to be distributed. The system also allows the user to identify (or modify the system selection) the tasks to be distributed. The tasks are transformed into autonomic elements by the system. The intention is not the development of the optimal distribution strategy for each application, but rather an acceptable (near optimal, but certainly better than a sequential system) distribution in terms of total elapsed time and throughput. It is necessary that the application demands are identified and resources utilized appropriately. Following is the outline of the goals of this system:

- *Autonomic environment:* The underlying system should be autonomic, i.e. it should support the properties of self-configuration, self-healing, self-optimization and self-protection. We believe that, only providing an autonomic environment to the application programmer is insufficient if building programs in such systems is not made simpler. If programming in such systems requires advanced knowledge of the workings of the underlying autonomic environment, then the goals of autonomic computing is not fulfilled. Users should program as usual with minimal constraints and the underlying system should transform the code in a way that could be self-managed autonomically.
- *Automatic distribution:* An average programmer wishing to utilize idle machines in the network does not want to take the responsibility for the physical distribution and coordination of objects. They are concerned more with the throughput and total execution time of the whole application. Therefore, programmers should need to do little (or nothing) to ensure the distribution of objects across the distributed environment. The system will automatically determine object dependability and allocate resources accordingly and coordinate the objects transparently at runtime. Advanced users can influence the system's automatic decisions by providing user specific policies. The system should hide the rest of the distribution and coordination process from the programmer. Programmers should have the impression that the objects are all executing on a

single machine, although in reality they may be executing somewhere else in the network.

- *Adding autonomic properties into existing code:* The goal is to inject autonomic primitives into non-autonomous system, whose source code is no longer available. Therefore the code must be analyzed and the autonomic functionality should be inserted in such a manner that it is separated from the service functionality of the existing code.
- *Ease of use:* The system should require minimal interaction from the programmer to achieve improved productivity and self-management. Problems encountered due to distribution should be hidden from the user and dealt with autonomically in the background. In doing so, the system should not jeopardize the user application or the integrity of the results. The system must provide an easy to operate user interface so that naive computer users can easily interact with the system and run their applications. On average, it should reduce the work and complexity associated with managing a large system. The system should be able to better respond to sudden changes in the environment and adjust its own settings appropriately.

Scope of This Research

The goal of this research is to develop an autonomic computing architecture to automatically perform the distribution of the user application across the available resources. This requires different issues [99] to be addressed. Addressing all of those issues and developing solutions for them in a single research project is somewhat impractical. Only the most important and essential issues are addressed and put forward to produce a robust and useful system within the time period of this research's life span. Further analysis and experimentation of other issues will continue as future work. Following is the scope of this research:

1. *Development of an autonomic architecture to support autonomic element execution*. This includes development of different repositories for autonomic systems, development of policy management scenarios, developing a self configurable and self optimizing framework for distribution.
2. *Automatic distribution of user program*s. This includes code transformation at byte code level and development of different design patterns for such transformations.
3. *Code transformation to inject autonomic primitives into the existing application*. The code needs to be analyzed and the autonomic functionality to be inserted in such a manner that it is separated from the service functionality of the existing code. However, transforming existing code to have all the different autonomic primitives is a daunting and time consuming task. Therefore, the thesis addresses self-healing primitive and associated transformations required to add this functionality into existing code.
4. *Formalize the creation and management of autonomic elements*. This includes, designing interfaces for user interaction with the system, and implementing policy management between autonomic elements.

This research has a broad scope that a single Ph.D. research project can not cover. Currently two Ph.D. researchers are working on different aspects towards the previously specified goals. However, more work needs to be done to develop a full-fledged system. The software architecture, code transformation techniques and algorithms that were developed by this research will assist in building such an extensive and complete autonomic system.

Challenges

Although nearly every aspect of autonomic computing offers significant engineering challenges [57, 83], the scope of this research has the following challenges to overcome:

1. There is no well designed software architecture for a distributed autonomic system. Designing and developing such software architecture is a substaential research challenge.
2. How should autonomic behaviors be injected with in an existing non-autonomous system? What if the source code for such system is unavailable and programmers has no knowledge of the code to include monitoring and actuating functions in the system.
3. The life cycle of an autonomic element (see Chapter 2 for definition) has several stages during its life time. Transition from one stage to the next seamlessly is a major challenge. How should the internal workings of an autonomic element be designed and modeled? What is a good architecture for an autonomic element? Furthermore, how should users interact with those elements? What would be a

good interface between the user and the autonomic elements or the autonomic system as a whole?

4. Program behavior is highly dynamic and a different distribution configuration may be appropriate in different phases of the execution of a program. As a consequence, program components should be able to relocate at runtime to enhance locality and communication benefits. What techniques are needed to seamlessly provide such support in a system?
5. How should such a system be evaluated? Should traditional benchmarking approaches with different functional data (execution time, throughput, latency etc.) be employed? Providing only such a functional evaluation of the system is not enough to judge its effectiveness in the field. Since one of the primary goals of this research is to provide an easy to use interface to the average user, an evaluation framework needs to be developed to address issues (openness, scalability, transparency, etc.) that can not be functionally evaluated.
6. Effectively partitioning a large computational problem and mapping and scheduling those partitions over the heterogeneous resources across the network is a substantial challenge.
7. Proper utilization of such a computing environment requires ways of estimating each components computation and communication needs and their dependencies so that an efficient mapping of components to resources can be achieved and the communication cost associated with the mapping is minimized.

Autonomic computing is a grand challenge for this century, where advances in several fields of science and technology (such as better machining learning techniques, knowledge representation and mathematical models for interaction among entities) are required to achieve the goals of autonomic computing. However, it is very important that the scientific and business community work towards the ultimate vision of autonomic computing and cooperate in building prototypes that quantifiably demonstrate the benefits of self-manageability.

Methodology

By defining a clear boundary of the problem and by having a detailed plan to solve the problem, the result of the research is likely to end up close to the goal. Although Autonomic Computing is still a new concept, the different technologies which are necessary to accomplish the vision of autonomic computing have mostly been developed over the past couple of decades. To improve our knowledge, existing research on the different facets of the autonomic computing paradigm which are researched independently and sparingly are studied. From this, generic theories can be setup and strategies to meet the goals set previously can be devised.

Java has been selected as the vehicle to demonstrate this thesis. The first step is to explore the functionality provided by the Sun Java API. In addition, the extensive range of middleware technologies is explored and compared to facilitate the development of the system and techniques for self-management. Since this research is performed at the byte-code level of the user code, it is then logical to explore all the tools that provide functionalities to work with byte codes. The exploration of such tools comprises technical literature survey, API evaluation and experimental coding. Furthermore, a comprehensive literature study was performed to try to understand the problem area.

Even though the approach presented in this thesis is well rooted as a method of research in computer science, we believe that if an application works in practice then the theories which it is built upon are valid. For this reason, we will be providing examples and experimented results that will show that our approach works in practice.

Contributions

The aim of this thesis is to explore self-managing software architectures, tools and techniques which hide programming and distribution complexities from the user. Towards this goal, the following original contributions are made by this thesis:

- A novel approach to autonomic management organization, autonomic element design and instrumentation of autonomic primitives into existing systems.
- A new approach to automatically transform an existing application into a self-managed application. This includes code transformation techniques to add autonomic primitives into existing applications. To perform such code transformations, requirements for autonomic computing are identified and solutions are provided for fulfilling those requirements.
- A software architecture to support self-managing distributed applications on heterogeneous platforms is developed. A three layered service-oriented peer-to-peer architecture is presented to provide self-management of user applications in a distributed environment.
- The design and development of basic autonomous entities is achieved to facilitate transparent self-management of systems.

- A prototype implementation of self-healing primitives into test applications is presented. The effects of such code transformations are analyzed and discussed.
- New software tools to improve a programmer's productivity in developing self-adaptive applications. The thesis also explores new software engineering approaches for emerging domains such as autonomic computing.
- The software tools explored in this thesis can be adapted to work for other experimental distributed systems. By further investigating into this work and improving the software tools we can get closer to fulfilling the vision of autonomic computing.
- A detailed survey of existing self-managing systems and approaches along with an intensive study of byte code manipulation tools and the shortcoming of existing middleware technologies in fulfilling the vision of autonomic computing is completed.

The research results of this thesis can benefit several areas of research and practice. Although the primary contributions of this research are in the area of self-adaptive distributed computing, some of the ideas explored by this thesis are applicable to other research areas, such as software engineering, programming languages, computer networking and operating systems.

Outline of the Thesis

The rest of the thesis is organized as follows: Chapter 2 discusses background information required to comprehend this thesis. This chapter gives a general overview of autonomic computing and explains issues relating to the research topic.

Chapter 3 presents a brief description of the Java Virtual Machine structure and Java's class file structure. This chapter also discusses Java byte code and the tools to manipulate byte codes.

Chapter 4 looks into other research projects that share similar goals to our research with respect to providing autonomic environments to users.

Chapter 5 presents the system architecture and discusses the code transformations required to achieve such self-managing properties in the system.

Chapter 6 presents the internal architecture of the autonomic element and provides an evaluation of the basic building block of any autonomic system.

Chapter 7 presents a prototype implementation of adding self-healing primitives into existing applications. The technique is evaluated and analyzed with different test applications.

Finally, Chapter 8 concludes this thesis and presents future work.

CHAPTER 2

BACKGROUND

This chapter introduces and discusses important background information necessary to comprehend this thesis and provides insight into the nature of autonomic computing systems.

What is Autonomic Computing?

The rapid upsurge in computational power over the past decades helped to produce ever more sophisticated software applications and environments, and fuelled the enormous growth in the number and variety of systems and components. As those systems become distributed and increasingly complex, software architects are less able to anticipate and design interactions among different components. Although there have been attempts to reduce such complexities within systems by introducing better software engineering practices (OOP, component based architecture etc.), the complexity remains as more and more new technologies and systems are being incorporated together. Such an environment is a complex, heterogeneous tangle of hardware, middleware and software from multiple vendors that are becoming increasingly difficult to program, integrate, install, configure, tune, and maintain. This leads to the idea of autonomic computing [42, 58] where the complexity and the management of such system is handled by the system itself.

This computing paradigm has been inspired by the human autonomic nervous system (ANS) [68]. ANS is the "*body's master controller that monitors changes inside* *and outside the body, integrates sensory inputs and effects appropriate response"* [77]. Examples of ANS's operations include, directing the heart to beat as a specific rate depending on body conditions; monitoring and correcting blood sugar and oxygen level; controlling eye pupils so that right amount of light enters the eye for reading; keeping one's body temperature close to 98.6 °F (37.0 °C) without any conscious effort and so on.

Autonomic computing is a new paradigm where computing systems possess the properties and capabilities of self-awareness and self-management. An autonomic computing system is an "Intelligent" open system that [35, 42, 58, 77]:

- "knows" itself,
- manages complexity,
- continuously tunes itself,
- adapts to unpredictable conditions,
- prevents and recovers from failures, and
- provides a safe environment.

There is an important distinction between autonomous activities in the human body and autonomic activities in an autonomic computing system. The decisions for many autonomic capabilities in the body are involuntary. On the contrary, autonomic capabilities in computer systems make decisions based on goals that the user sets for the system using policies. Usually these are adaptive policies, rather than hard-coded procedures, which determine the types of decisions and actions that autonomic capabilities should perform.

For any autonomic application or system, the autonomic element is the fundamental atom, which is a modular unit of composition with pre-defined interfaces and mechanisms for self-manageability. As shown in Figure 1, an autonomic element typically consists of one managed element, which could be any computing entity that requires self-management. The autonomic element interacts with other elements and programmers via their autonomic managers. Each autonomic element is responsible for managing its own internal state and behavior and for managing its interactions with other autonomic elements and the environment in which it is residing. The internal behavior of any autonomic element and its interaction with other such elements is governed by the goal set by the developer of that element. The elements may be required to help other elements to achieve their goals. As it can be seen from Figure 1, each autonomic manager has four distinct functional component devoted to individual functionality of the autonomic element.

- The *monitor* component monitors the managed element and itself and provides that data to the *analyze* part.
- The *analyze* component takes data from the monitor and analyzes the current operation with respect to the set policies.
- If any changes to be made in the behavior of the autonomic element, the *plan* component assigns task/resources based on policies and performs policy management.
- Once a new plan is devised, the *execute* component performs that operation on the autonomic element or on the managed element.

There are *sensors* and *effectors* in the autonomic elements, which are interfaces either between the autonomic manager and the environment or between autonomic manager and

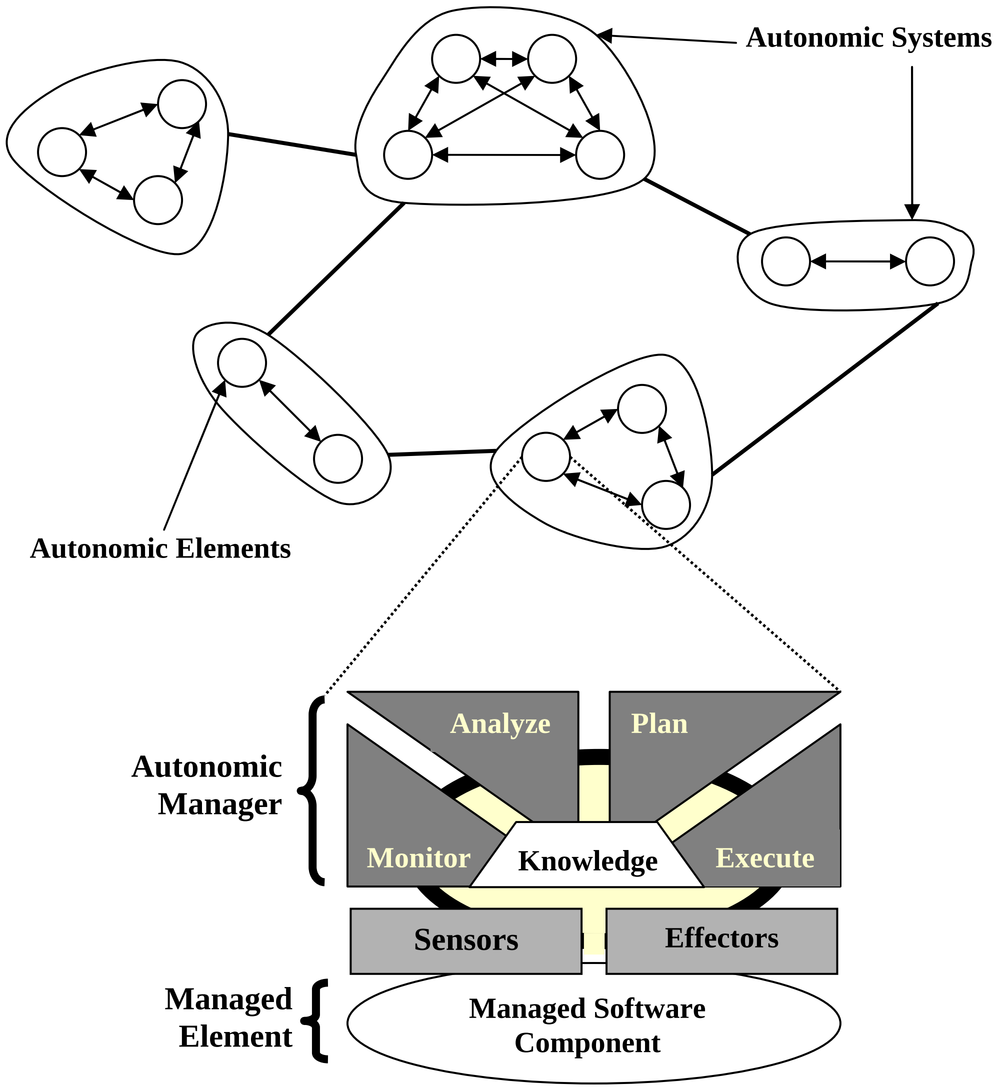


Figure 1. Structure of an Autonomic Element.

the managed element. The *sensors* help the autonomic manager to monitor and sense the environment or the managed element, whereas the *effectors* helps propagate external policies to an autonomic element or send control information to a managed element.

As shown in Figure 1, an autonomic system is composed of autonomic elements, which are capable of managing the system's behavior as a whole and may engage in relationships with other autonomic systems in accordance with high level policies. Any autonomic system should exhibit the following major characteristics [58, 67]:

- *Self-configuring*: An autonomic system must be able to install and establish itself automatically across the available resources. In addition, such systems should adapt automatically and dynamically to environment changes.
- *Self-healing*: An autonomic system must be able to detect any faults and runtime anomalies and should recover from that and continue running smoothly. The main objective of self-healing is to maximize availability, survivability, maintainability and reliability of the system [36]. This, however, does not include logic errors and poor programming on behalf of the application developer.
- *Self-optimizing*: An autonomic system must be able to detect any sub-optimal behaviors and optimize itself to improve its execution [77]. Autonomic elements attempt to optimize resource allocation and in turn will try to maximize their utilization for satisfying goals set by the user.
- *Self-protecting*: Autonomic systems must detect and protect computing assets from both internal and external threats. Autonomic systems must maintain integrity and accuracy of the system and should administer the overall system security.

Along with the above described major characteristics, autonomic systems also have the following minor characteristics [77, 83]:

- *Self-awareness*: An autonomic system should know its current state and behavior, so that it can collaborate with other autonomic systems.
- *Context-awareness*: An autonomic system should be aware of its environment and can react to any change in its environment.

- *Anticipatory*: An autonomic system should be able to anticipate to any extent possible, changes in the state of the system and should be able to manage itself proactively.
- *Open*: An autonomic system should be portable across heterogeneous computing environments and should be implemented with open standards and protocols.

Why Java?

The current trends in computer software is that it is network centric, comprehensive, heavily distributed, mobile, highly integrated and incorporated with just about everything. Java presents features, such as platform-independence, easy but powerful programming concepts and transparent network integration. This research employs Java [95] because of its features, ease of programming and its ability to meet the demands of future software developments.

Java and Java Remote Method Invocation-Over Internet Inter-ORB (Object Request Broker) Protocol (RMI-IIOP) [96] is used for implementing the framework and algorithms. Although Remote Procedure Call (RPC) [8] and Message Passing Interface (MPI) [63] are well known and well-understood mechanisms for making distributed computing look like centralized computing, both of them have severe limitations [34] which hinders the goals of this project. However, the distributed object model, with the help of an object oriented programming language such as Java, eliminates several of these limitations and simplifies the function of a distributed system. In Java RMI, from the viewpoint of a process, a distributed object appears to be the same as a local object. Using an object centric communication protocol permits the use of standard web centric communication protocols, such as WSDL [105] or SOAP [91], in the future. The Java RMI environment greatly simplifies distributed object programming by allowing communication between two remote objects with minimum constraints.

Java RMI enables Java programmers to distribute computation across a networked environment without having to worry about low-level networking details. Since RMI is central to Java, it brings the power of Java's safety and portability to distributed computing. Java includes features which serve as important building blocks for implementing the proposed system:

- *Reflection*: Java provides reflection that allows objects to determine type information from an object instance or to create an object instance from type information. Reflection allows instances of objects to be created even if the class name is not known until run-time. Reflection can also be used to determine the methods and fields of objects that are dynamically created. Reflection is needed to gather user program's information and during dynamic class loading in the autonomic elements.
- *Object Serialization*: Java serialization converts objects into a format where they may be stored as a sequence of bytes. Serialization is vital for the transfer of objects between autonomic elements. Serialization occurs transparently in Java when an object is passed as a parameter in a remote method invocation or when objects are passed through I/O streams.
- *Dynamic class loading*: Java supports dynamic class loading, which allows code to be loaded only when required. Dynamic class loading implies the ability to reference and instantiate objects that are unknown to the application at compile

time. This enables an autonomic element to keep executing and requesting remote object, without having any prior knowledge about the objects to be created.

- *Security manager*: The Java security manager is a class that allows application to implement a security policy. A security policy defines those Java features that objects are allowed to access. The security manager allows the system to ensure security for those who provide computing resources, by restricting object access to those API features, which do not affect the remote host.

With the above features, RMI provides a simple yet direct model for distributed computation with Java objects. RMI allows calls to be made between Java objects in different virtual machines or even on different physical machines. Coupled with the Java platform's code portability, RMI greatly simplifies distributed object programming. Primary advantages of RMI are [109]:

- *Object oriented*: RMI can pass full objects as arguments and return values, not just pre-defined data types. In RPC systems, peers have to decompose such an object into primitive data types, ship those data types and reconstruct the object back on the other side.
- *Write Once, Run Anywhere*: RMI is part of Java's "*Write Once, Run Anywhere*" approach. Any RMI-based system is totally portable to any Java Virtual Machine (JVM).
- *Design patterns*: RMI provides the full power of object oriented technology in distributed computing. This opens up new opportunities to use object oriented design patterns to simplify the system design [59].

- *Secure, portable applications*: RMI preserves Java Runtime Safety, enabling developers to write distributed applications that are both secure and portable.
- *Parallel Computing*: RMI is multi-threaded that allows the autonomic elements to exploit Java threads for better concurrent processing of managed elements.
- *Ease of use*: RMI is easy to write and easy to use. RMI makes it simple to write remote Java servers and Java clients that access those servers.

RMI provides a solid background for object oriented distributed computing. RMI which uses Internet Inter-ORB Protocol (IIOP) [96] as its transport protocol is chosen as the primary communication method instead of standard RMI over JRMP (Java Remote Messaging Protocol). RMI-IIOP comes with the Object Management Group's (OMG) Interface Definition Language (IDL) [47] (Sun Version) through which programs written in other languages can also be connected and form part of the application or system. RMI-IIOP is a robust choice for a dynamic environment as it is necessary to implement the autonomic elements in a manner that it follows different industry standards for interfacing and internetworking. RMI-IIOP permits the extension of the benefits of RMI with new standards and functionalities.

Although, the target application should be written in Java; the proposed approach could also be implemented in other Common Language Runtime [66] based interpreted languages, such as C#. However, currently this avenue is not being explored as it constraints the approach to a machine dependent environment. Hopefully with improvements in inter-system interfaces, such as Mono [80] and Wine [108], this work can be extended in C# in the future.

Which JVM?

Although the standard JVM developed by Sun Microsystems provides different functionalities for distributed programming and for developing autonomic elements, it is limited to a certain extent when it comes to interaction with the environment. One solution to this limited functionality is to modify the JVM and tailor it to meet the needs of this research. There are several modified JVMs [5, 9, 21, 25] available which provide extensive functionalities to interact with the underlying machine. However, using a modified virtual machine is not an acceptable solution in all situations as it would require the modified JVM to run on all nodes of a distributed application and, therefore, disable the use of a standard JVM. Furthermore, the modified JVMs are typically not tested well as compared to the standard JVM. This may have a negative impact on performance.

Currently this research uses native methods for machine dependent interaction needed by the autonomic elements. The autonomic elements need to interact with the environment for several reasons, such as, measuring machine and networking load, bootstrapping autonomic elements and migration of autonomic elements. Currently the standard support provided by the Java API is used for these operations. Native methods will be implemented in the future for these operations on different platforms for extensive support needed for those operations. The transformed existing user code uses only Java and RMI-IIOP and does not require any changes to the JVM to perform object manupilation and environment interaction in different hosts.

Source Code or Byte Code?

The motivation for choosing byte code level modification comes from the fact that existing systems usually do not have the source code for modification. In those scenarios, it is beneficial if one could work with byte code and perform associated transformations necessary to run that program autonomically. Furthermore, the byte code is already free of compilation errors and optimized for execution. Byte code also follows a strict format compared to corresponding source code version of the program. This makes it easy to manipulate and to work with byte codes. If users wish to automate a new program, which has its source code available, the presented approach still works as the only thing that needs to be done is to simply compile the source code with the standard Java compiler and then input it to the presented system. This factor gives an extra edge to the approach presented in this thesis, as a source code pre-processor with extra functionality that assists application programmers can be developed to add new functionalities (as meta-programming) in a new program and the pre-processor simply converts that to the corresponding Java source code, which in turn can be provided to the system after compilation and conversion to byte code.

Summary

Autonomic computing is a new paradigm in computing, which aims to produce software that has the properties of self-configuration, self-healing, self-optimization and self-protection. This chapter presents the basic concepts of autonomic computing and describes different aspects of it. Java has been chosen for this research mainly because of its features, ease of programming and its ability to meet the demands of future software developments. Different features of Java, which is important to this research, are discussed in detail. Since the Java Virtual Machine (JVM) specification is implemented by multiple parties, the selection of a particular JVM implementation for this research is important. Although modified JVMs provided some extended functionalities, the standard JVM developed by Sun Microsystems is the most widely used in industry and academia. For this reason, this research chooses to use Sun JVM for wider acceptance in future. Finally, reasons for working at byte code level are explained and discussed.

CHAPTER 3

JVM AND JAVA CLASS FILE

This chapter provides an introduction to the Java Virtual Machine (JVM) [50] which is an abstract computer where all Java programs execute. Later in the chapter, a detailed discussion of Java's class file is provided; this is the runtime representation of a class from the source code.

Structure of the JVM

The Java Virtual Machine (JVM) is a specification of an abstract computer to run Java programs. The specification defines a set of features that every Java Virtual Machine must have. However, the implementation is left to the developers. The specification is flexible enough to allow a Java virtual machine to be implemented either completely in software or to varying degrees in hardware. The flexible nature of the specification enables the JVM to be implemented on a wide variety of devices. The main job of a JVM is to load class files and execute the byte code [37, 110] inside them. As shown in Figure 2, the *class loader* inside the JVM loads class files for both the running program and any system classes that the program utilizes. The *execution engine* is responsible for the execution of the byte code inside those classes. A Java program usually interacts with the host operating system by invoking native methods. Java has two kinds of methods: Java methods and native methods. A normal Java method is written in Java, compiled to byte code, and then stored in class files. On the other hand, a native method is written in some other programming language and compiled to the underlying machine code of a

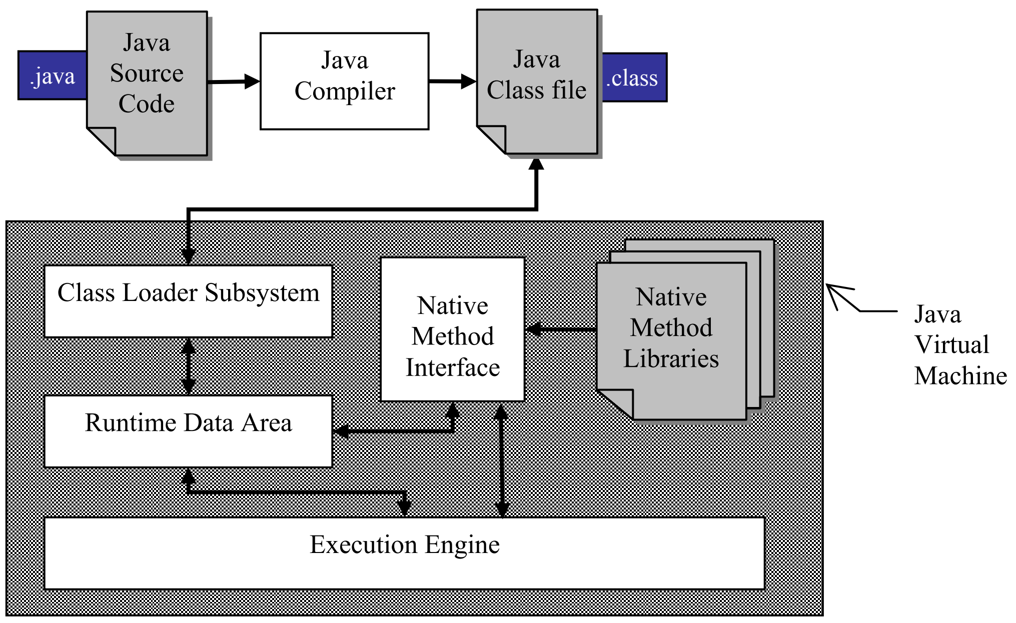


Figure 2. Structure of the Java Virtual Machine.

particular processor. Usually native methods are stored in a platform specific dynamically linked library. Native methods are the connection between a Java program and an underlying operating system. The *runtime data* area of the JVM holds all the necessary runtime data of a running Java program. The runtime data area consists of the following:

- *Method area*: This area holds runtime information of the class file. The method area is shared among all threads of that class. It contains static class information such as field and method data, the code for the methods and the constant pool. The constant pool is a per-class table, containing various kinds of constants similar to a symbol table.
- *Heap*: The heap is the data area where all objects and arrays are allocated. The heap is shared among all threads. As a program executes, all the objects that the program instantiates are placed on the heap.

- *PC register*: The program counter (PC) register indicates the next byte code instruction to be executed.
- *Stack*: Each thread has a private stack area that is created at the same time as the thread. The Java stack is composed of *stack frames*. A stack frame contains the state of one Java method invocation. When a thread invokes a method, the Java virtual machine pushes a new frame onto that thread's Java stack. When the method completes, the virtual machine pops and discards the frame for that method.

Figure 3 shows a snapshot of the runtime data area of the JVM at a particular moment. In this particular situation, four individual classes are loaded in the method area. The method area and the heap are shared among all the threads running inside that particular instance of the JVM. As the program executes, all objects that the program

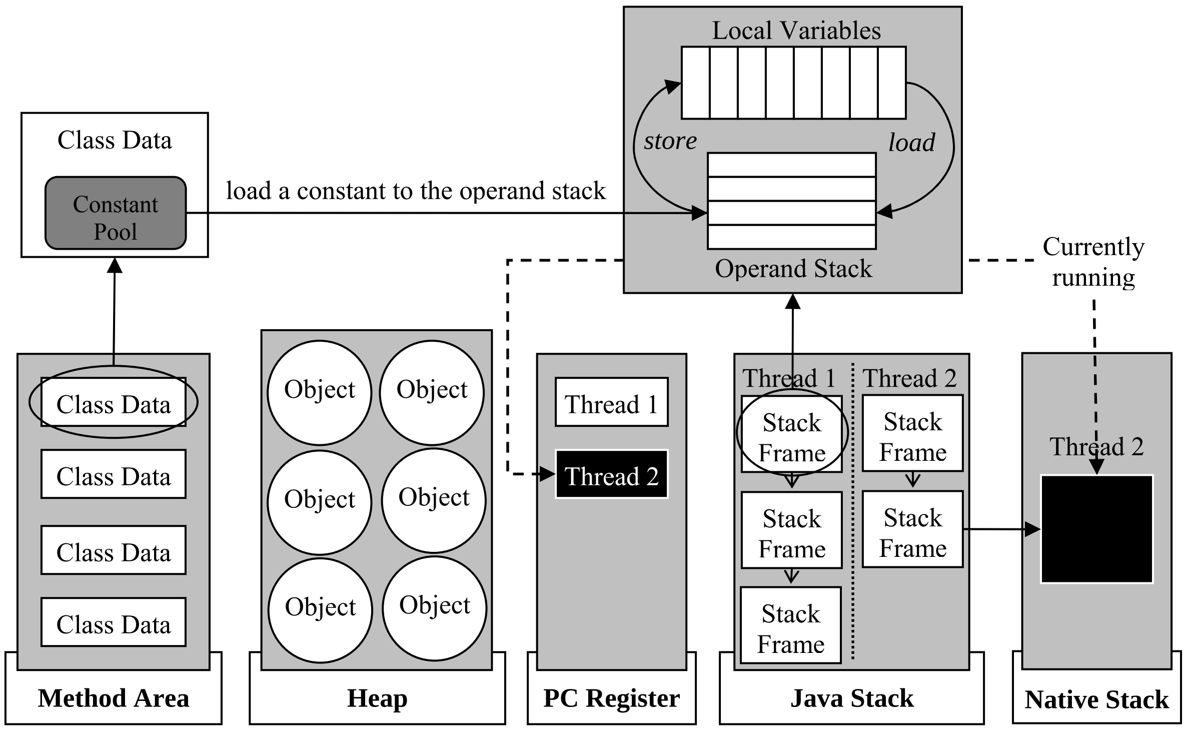


Figure 3. JVM Runtime Data Area.

instantiates are placed on the heap. In Figure 3, five individual objects of different classes have been created. As additional new threads are created, each thread receives its own program counter (PC) and stack area. If the thread is executing a Java method, the PC register indicates the next byte code instruction to execute. For native methods, this information is either stored in the PC or an implementation dependent memory area. The JVM has no registers (analogues to registers inside any modern processor) to hold intermediate data values. However, the operand stack replaces the registers in a conventional processor.

Data Types and Values

The type of a variable or expression in Java is known at compile time. In doing so, much of the runtime type checking is avoided. The types in JVM are classified into following two groups:

1. Primitive types: This data type represents different types of numeric values and can only hold values of any one specific type. Primitive data types are always passed by value.
2. Reference types: This type represents complex data such as *interface, class, array* and *null* (reference to nothing) pointer. Reference types pass values indirectly.

Variables

There are several different kinds of variables in JVM as shown in Table 1. Depending on the scope and context of the variable, it has to be handled accordingly in byte code level.

Table 1. Variables inside JVM.

Variable Type	**Description**
Class	Created on a per class basis.
Instance	Created on a per instance basis.
Array component	Created for each array element. However, each of these is unnamed and can not be accessed directly.
Method parameter	Temporary variable to hold an argument passed to a method.
Constructor parameter	Same as a method parameter, but used for constructor arguments.
Exception handler parameter	Temporary variable to catch any exception that has been thrown.
Local	Temporary variable that exists during the execution of a statement or block of statements.

Fields and Methods

Each class in Java consists of two parts, fields and methods. Fields are the variables defined outside of any method that hold the properties of that class, whereas the methods are operations through which those properties are manipulated. The *constantpool* of the class file, which holds information for each of the fields and methods, is discussed in the next section.

Structure of Java Class File

The Java class file is the compiled code (byte-code) corresponding to the source code, which is ready to be executed by the JVM and is represented using a platform independent binary format. A Java class file is defined as an array of bytes with a format defined by Sun Microsystems [50]. Figure 4 shows the overall structure of the class file [37, 110]. The first eight bytes of the class file hold the header information. Of these eight bytes, four bytes denotes the magic number, which makes non-Java class files

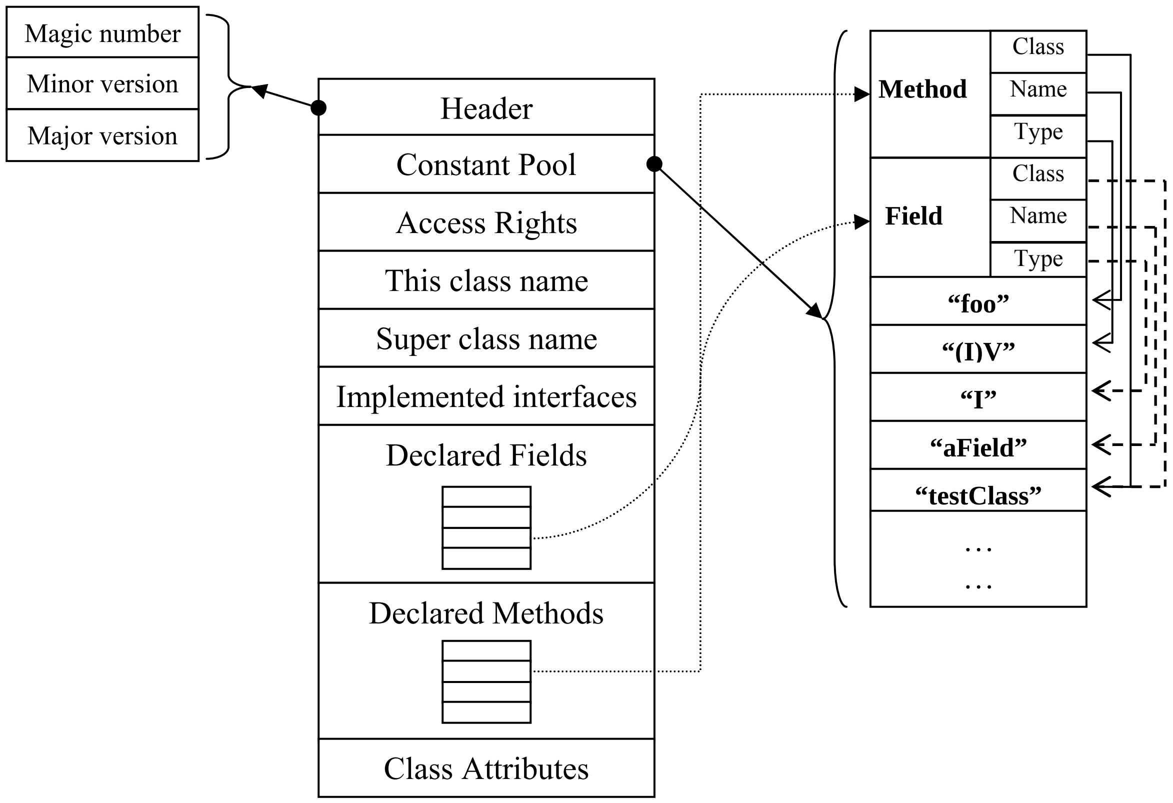


Figure 4. Structure of a Java Class File.

easier to identify. The next four bytes of the header contains versioning information related to the Java technology development. The constant pool is a heterogeneous array of constants about different components of the class file. It occupies the most space in any class file. The constant pool is organized as a list of entries. Entries in the constant pool are referenced by the corresponding element's index, where the index starts from 1. Many entries in the constant pool refer to other entries in the constant pool. Table 2 shows the different types of element inside the constant pool and what they represent.

The next two bytes after the constant pool is an encoded bit mask, which represents access privileges about the class and any interfaces being implemented by that

Table 2. Constant Pool Entries.

Constant Pool Entry	**Representing**	**Description**
CONSTANT_Class	A Class or Interface	Fully qualified name.
CONSTANT_Fieldref	A Field	Fully qualified name of the class containing the field and the descriptor of the field.
CONSTANT_Methodref	A Method in an object	Fully qualified name of the class containing the method and the descriptor of the method.
CONSTANT_InterfaceMethodref	A Method in an interface	Fully qualified name of the interface containing the method and the descriptor of the method.
CONSTANT_String	Sequence of characters	Constant objects of type String.
CONSTANT_Integer	An integer	
CONSTANT_Float	A float	
CONSTANT_Long	A long	
CONSTANT_Double	A double	
CONSTANT_NameAndType	Name and type information of a method or a field	Method or Field name and descriptor.
CONSTANT_Utf8	Sequence of characters	String values.

class. The next two bytes represent the name of the current class and its super class. These are actually indices into the constant pool, where the entry at that position in the constant pool is a *CONSTANT_Class* entry, representing the corresponding classes. The next information in the class file represents the number of interfaces the class is implementing and corresponding indices of those interfaces in the constant pool table. Following this information, a list of fields and methods declared in the class is represented in the same way as the interfaces are represented.

The last components in the class file are the attributes, which give general information about a particular class or interface defined in the class. Attributes come in various forms and are generally defined by the JVM specification. However, programmers can create their own attributes and add them to the class file. The JVM will ignore those user defined attributes at runtime. Furthermore, attributes can appear in several places in the class file, not only at the top level class file table (Figure 4). Attributes can have other attributes nested within them, representing different aspects of that particular attribute. The attributes that appear in the class file table (Figure 4), provide more information about that class or the interfaces that the class is implementing. Similarly, attributes that give more information about a field are attached to the fields, and attributes that give more information about a method are attached to the methods. Table 3 shows the attributes that are defined by the JVM specification. Since this thesis is focused on modifying the runtime behavior of a method, following is the description of some of the most important attributes, which are attached to the methods and its byte code instructions.

Table 3. Java Class Files Attributes.

Attribute Name	**Represent**	**Where used**
ConstantValue	Value of a static constant field	Field
Code	JVM instructions	Method
Exceptions	The exceptions a method may throw	Method
InnerClasses	Any inner classes	Class file
Synthetic	A class member that does not appear in the source code	Class file, Field and Method
SourceFile	Name of the source code file	Class file
LineNumberTable	A mapping between source code line number and byte code instructions	Code Attribute
LocalVariableTable	A mapping between the representation of local variable (a number) and the source code representation (a name)	Code Attribute
Depricated	A class, interface, field or method has been superseded	Class file, Field and Method

Code Attribute

The variable length code attribute contains the JVM instructions and auxiliary information for a method. The code attribute holds the body of the method it is associated with. Since the JVM is a stack based interpreter, the code attribute defines the maximum stack depth that may be reached while executing its method body. It also defines the maximum number of local variables being used. An array inside the code attribute holds the actual byte code instruction sequence. The code attribute also defines an exception table. The exception table holds the information related to each of the try-catch exception handling blocks in the code attribute.

Local Variable Attribute

This is an optional, variable-length attribute, which may be used by debuggers to determine the value of a given local variable during the execution of a method. Usually, the Java compiler does not write this attribute during compilation. Only when the debugging switch is used during compilation (*javac –g*), does Java add this attribute to each method's code attribute. Each local variable that appears in the code attribute is represented by its scope information and an index in the constant pool defining its type.

Line Number Attribute

This variable length attribute provides a mapping between the byte code offset to line numbers in the source file. The entries in this table may appear in any order.

Exception

This variable length attribute lists the checked exceptions that a method may throw. This is an array of indexes into the constant pool entries for the exceptions declared in this method's *throw* clause.

Byte Code Instructions

The instruction set of the JVM contains 201 different instructions [50]. Most of these instructions in the JVM instruction set explicitly encode type information about the operations they perform. For instance, opcodes for integer type start with 'i', opcodes for floating point type start with 'f' and so on. All of these byte codes can be grouped into the following categories:

- *Load and store*: Load instructions push values from the local variable table onto the operand stack. On the other hand, store instructions transfer values from the stack back to the local variable table. 70 different instructions belong to this category. Short versions (single byte) exist to access the first four local variables in the stack. There are unique instructions for each basic type (*int*, *long*, *float*, *double* and *reference*). This differentiation is necessary for the byte code verifier, but is not needed during execution. For example *iload* (load an integer), *fload* (load a float) and *aload* (load an object) all transfer one 32-bit word from a local variable to the operand stack.
- *Arithmetic*: This type of instruction operates on the values found on the operand stack and pushes the result back onto the operand stack. There are arithmetic instructions for *int*, *float* and *double*. There is no direct support for *byte*, *short* or *char* types. These types are converted back and forth to integers and manipulated as

integers. An example of these instructions is, *iadd*, to add top two stack elements (which are integers) and put the results back to the operand stack.

- *Type conversion*: The type conversion instructions perform numerical conversions between all Java types, as implicit widening conversions (e.g. *int* to *long*, *float* or *double*) or explicit (by casting to a type) narrowing conversions. For instance, *i2b* will convert an integer value to a byte value.
- *Object creation and manipulation*: Class instances and arrays (which are also objects) are created and manipulated with these instructions. Objects and class fields are accessed with type-less instructions. For example, *new* will create a new class instance.
- *Operand stack manipulation*: All direct stack manipulation instructions are type-less and operate on 32-bit or 64-bit entities on the stack. Examples of these instructions are *dup*, to duplicate the top operand stack value, and *pop*, to remove the top operand stack value.
- *Control transfer*: Conditional and unconditional branches cause the JVM to continue execution with an instruction other than the one immediately following the current instruction. Branch target addresses are specified relative to the current address with a signed 16-bit offset. The JVM provides a complete set of branch conditions for *int* values and references. A conditional branch based on a comparison between data of types *long*, *float*, or *double* is initiated using an instruction that compares the data and produces an *int* result of the comparison. A subsequent *int* comparison instruction tests this result and affects the original conditional branch.

- *Method invocation and return*: The different types of methods are supported by four instructions: invoke a class method (*invokestatic*), invoke an instance method (*invokevirtual*), invoke a method that implements an interface (*invokeinterface*) and an *invokespecial* for an instance method that requires special handling, such as private methods or a superclass method.

Other than these, there are a few instructions for throwing exceptions or for synchronization or to implement Java's *finally* construct. Usually all byte code consists of one instruction byte followed by optional operand bytes. The length of the operand is one or two bytes, with the following exceptions:

1. *multianewarray* contains 3 operand bytes,
2. *invokeinterface* contains 4 operand bytes, where one is redundant and one is always zero,
3. *lookupswitch* and *tableswitch* (used to implement the Java switch statement) are variable length instructions, and
4. *goto_w* and *jsr_w* are followed by a 4 byte branch offset, but neither is used in practice as other factors limit the method size to 65535 (2^8) bytes.

An Example

In this section, through a simple example, the class file format is illustrated. The *testClass* program, whose source code is presented in Figure 5, is discussed at the byte code level. The example program has a single field and method, where the method only prints some information on the standard output device. This example program presents a

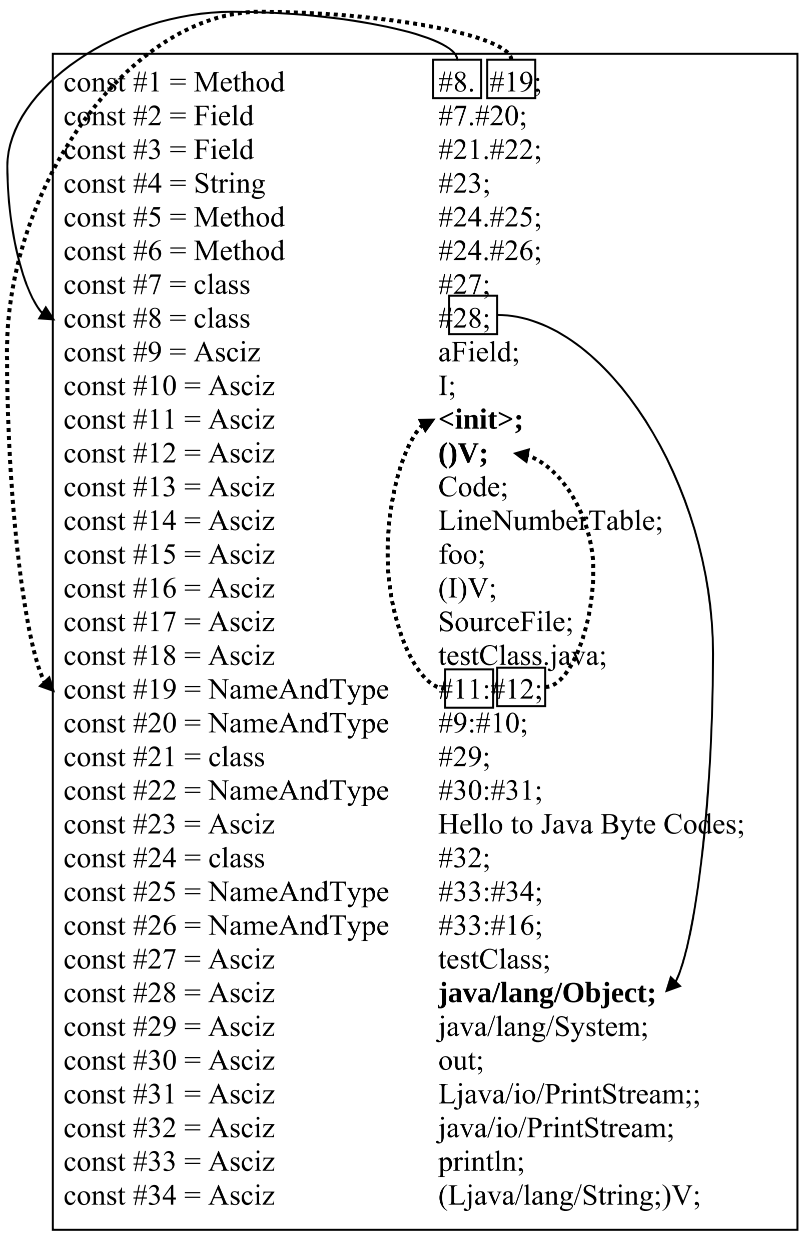
```
public class testClass{
    int aField=10;

    public void foo (int num){
        System.out.println("Hello to Java Byte Codes");
        System.out.println(num);
    }
}
```

Figure 5. An Example Java Class.

simplified view of the class file and associated components and it does not cover all the Java semantic elements.

Once the program is compiled with the Java compiler (*javac testClass.java*), the supplied byte code analyzer (*javap*) can be used to examine the content of the class file. Figure 6 shows the constant pool structure of the class file. The compiler automatically generates a zero argument constructor which calls the default constructor of the class *Object*. As shown in Figure 6, this newly added constructor information is illustrated with the help of arrows. So, entry 1 of the constant pool indicates that it is a method which is of the class identified at index 8 and this method's name and type information is at index 19. This process repeats and the method descriptor for this method is determined from the constant pool. For the example entry (entry 1), the method is of *java.lang.Object* class, its name is <*init*>, and has no argument (*()*) and returns nothing (*V* for *void*).

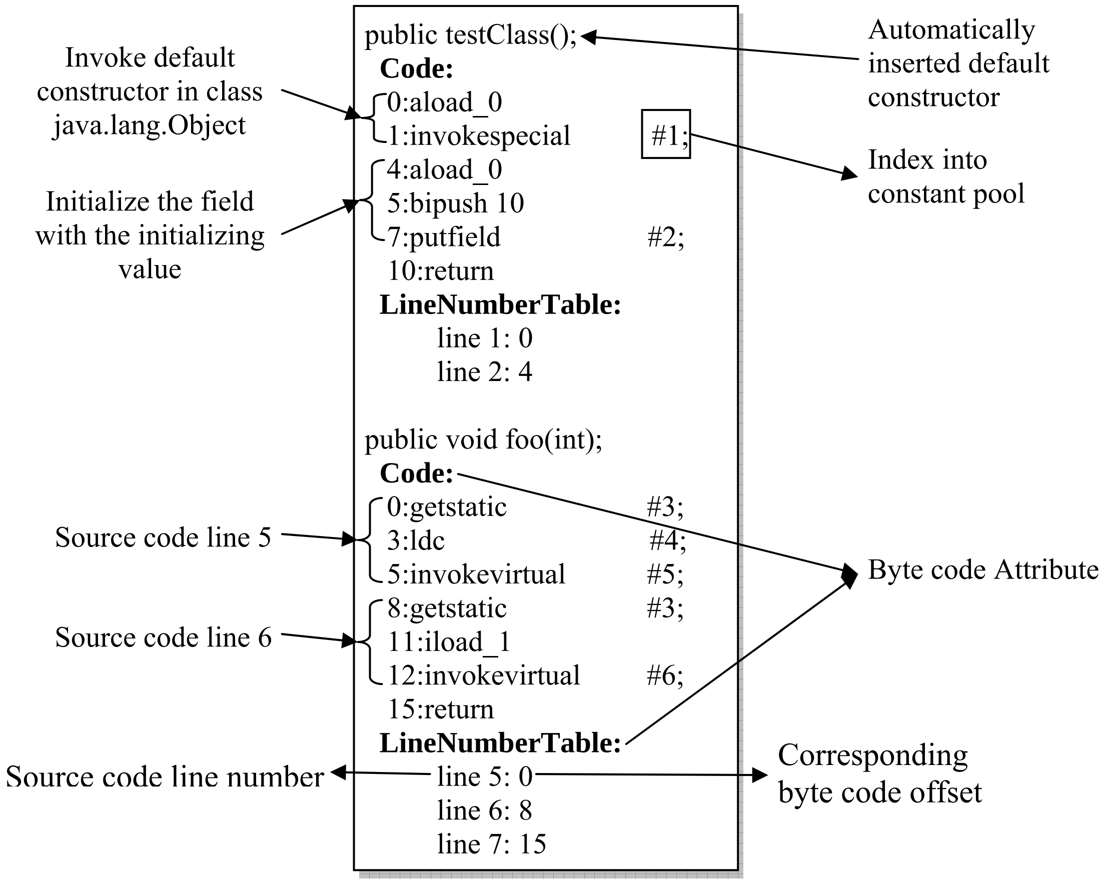
The method structure for the given class is shown in Figure 7. As described earlier, the body of the automatically inserted default constructor is also shown. The default constructor first invokes the default constructor of the class *java.lang.Object* and



```
const #1 = Method            #8.#19;
const #2 = Field             #7.#20;
const #3 = Field             #21.#22;
const #4 = String            #23;
const #5 = Method            #24.#25;
const #6 = Method            #24.#26;
const #7 = class             #27;
const #8 = class             #28;
const #9 = Asciz             aField;
const #10 = Asciz            I;
const #11 = Asciz            <init>;
const #12 = Asciz            ()V;
const #13 = Asciz            Code;
const #14 = Asciz            LineNumberTable;
const #15 = Asciz            foo;
const #16 = Asciz            (I)V;
const #17 = Asciz            SourceFile;
const #18 = Asciz            testClass.java;
const #19 = NameAndType      #11:#12;
const #20 = NameAndType      #9:#10;
const #21 = class            #29;
const #22 = NameAndType      #30:#31;
const #23 = Asciz            Hello to Java Byte Codes;
const #24 = class            #32;
const #25 = NameAndType      #33:#34;
const #26 = NameAndType      #33:#16;
const #27 = Asciz            testClass;
const #28 = Asciz            java/lang/Object;
const #29 = Asciz            java/lang/System;
const #30 = Asciz            out;
const #31 = Asciz            Ljava/io/PrintStream;;
const #32 = Asciz            java/io/PrintStream;
const #33 = Asciz            println;
const #34 = Asciz            (Ljava/lang/String;)V;
```

Figure 6. Constant Pool of the Given Class.

then initializes any fields in the class. Byte code instructions point to constant pool entry by using the corresponding constant pool index number. *LineNumberTable* is used to map between the source code line and byte code offset for debugging purposes.



```
public testClass();
 Code:
  0:aload_0
  1:invokespecial        #1;
  4:aload_0
  5:bipush 10
  7:putfield             #2;
  10:return
 LineNumberTable:
     line 1: 0
     line 2: 4

public void foo(int);
 Code:
  0:getstatic            #3;
  3:ldc                  #4;
  5:invokevirtual        #5;
  8:getstatic            #3;
  11:iload_1
  12:invokevirtual       #6;
  15:return
 LineNumberTable:
     line 5: 0
     line 6: 8
     line 7: 15
```

Invoke default constructor in class java.lang.Object
Initialize the field with the initializing value
Source code line 5
Source code line 6
Source code line number
Automatically inserted default constructor
Index into constant pool
Byte code Attribute
Corresponding byte code offset

Figure 7. Byte Code Instructions Inside the Methods.

Byte Code Modification Tools

It is important to survey all necessary tools that manipulate byte code and that could fulfill the needs of the system being developed and associated code transformations. It is essential to ensure that there are no byte code manipulation tools already exists which offer the same set of features that are presented in this thesis. Furthermore it becomes important to find the most extensive byte code manipulation tool that may help develop the system advocated in this thesis. Moreover developing a tool that could simplify the abstractions presented in the last few sections is a challenging task. To avoid reinventing the wheel, it is preferable to use third party tools already in existence and utilize them in the development of the proposed system. From this viewpoint, this section is an integral part of this thesis, which gives a glimpse into the tools that were surveyed and evaluated. Although there are several byte code manipulation tools [11, 12, 15, 17, 90, 92], the behavior sought from the tools available is discussed first.

The choice between the different byte code manipulation tools is governed by the following issues:

1. *Expressiveness*: How well the tool can express the constraints appearing in the class file, which are described by the JVM specification [50]. For example, a particular tool, which is not able to realize the exception table, will prevent faithful handling of the exceptions mechanism used in Java source code to handle errors.
2. *Maturity*: How long the tool has being used by others and whether any substantial analysis of that tool has ever been done and a comparison to other such tools performed? How many different systems were developed using a particular tool? If the selected tool is well used and tested and free of any software bugs, it is unlikely to have runtime anomalies due to the use of that tool for dynamic modification of the byte code.
3. *Programming language in which it is implemented*: Since the tool is required at runtime for dynamic reflection and modification of byte codes, it is desired that the tool itself is built in Java. To achieve the goal of portability, it is also desirable

that the tool is shipped as portable packages and is not tied to any other packages other than Java system packages.

4. *The quality of the abstractions provided*: What level of abstraction do the tools provide? Although a higher level of abstraction is required for achieving the proposed approach, sometimes lower level manipulation primitives are also necessary. A tool that provides both levels of abstraction is useful. For instance, a low-level abstraction may allow the addition of a byte code instruction by using an index in the constant pool after checking that this index is not outside of the constant pool. A higher level abstraction may ease that programming task by hiding all the low level details and by automatically maintaining the constraints imposed by the JVM specification.
5. *Open source*: Does one need to worry about any licensing issues? Is the tool completely open source? Can the tool be modified to suit the goals of this thesis?
6. *Documentation/help provided*: How much help is provided by a certain tool? Are there well developed manuals and examples to use that particular tool? Do the developers provide any sort of support for using their tool?

There are several general-purpose implementations of byte code manipulation tools available. Some are developed to assist with Aspect-oriented programming, while others translate the class file format into an internal representation and then allow the programmer to work on it. After extensive surveying, the following two byte-code manipulation tools were selected as candidates for consideration. Both of the following fulfill the above described selection criteria:

- *BCEL*: As described in [20], the purpose of BCEL (Byte Code Engineering Library) [12] is "to give the users a convenient possibility to analyze, create, and manipulate (binary) Java class files". BCEL provides two different representations of the class file: a static one and a dynamic one. Each representation has an associated package in BCEL that clearly separates the scope of the different abstractions in use. The static level is used to describe a class from a virtual machine point of view while the dynamic level allows actual modifications on a class file. The static level acts as an intermediate representation; using the dynamic level requires building the abstractions representing a given class at a static level first. Saving a dynamically modified class file requires the regeneration of a static description from the dynamic description. This approach makes it rather cumbersome to use and BCEL's method of improving the abstraction relies upon the use of programmer's object-oriented techniques and knowledge of design patterns. Although BCEL has been used to develop several systems, it was not selected for use due to its steep learning curve and awkwardness of use.
- *Javassist*: Javassist (Java programming assistant) [15] is a load-time reflective system for Java. It is a class library for editing byte codes in Java. It provides source-level and byte-code level abstractions. Javassist enables Java programs to define a new class at runtime and to modify a class file even before the JVM loads it. Unlike other similar systems, Javassist provides source-level abstractions which allow programmers to modify a class file without detailed knowledge of the Java byte code. Javassist comes with a built in compiler that can compile a

fragment of source text and insert the corresponding byte code inside an existing byte code sequence. This ease of use is a unique feature of Javassist compared to other tools and is a key element in choosing Javassist as a byte-code manipulation tool for this research.

Summary

Since the research work with byte-code and run-time code transformations, it is important to have a clear idea of internal workings of the Java Virtual Machine and Java Class file. This chapter therefore gives a brief introduction of both the JVM and the structure of the Java class file. Since this research is working at the byte code level, discussion of byte-code instructions is then followed with examples to illustrate different aspects of class structure and byte-code instruction format. There are a number of tools available to manipulate byte codes. It is important to survey all necessary tools that could manipulate byte codes and that could fulfill the needs of the system being developed and associated code transformations. Therefore requirements are identified and available tools are surveyed to see whether they fulfill the requirements being set.

CHAPTER 4

RELATED RESEARCH

This thesis investigates new software tools for automating and separating the distribution concerns in programming distributed systems. This work is placed at the intersection of automatic program partitioning, automatic program distribution, autonomic system management, software engineering and distributed systems. The motivation of this research comes from the fact that it is inherently difficult to automate the process of distributing an existing centralized system over a set of distributed heterogeneous machines. Current related research focuses on a single issue at a time. Typically, the options explored are not transparent to the user and, in fact, require the user to be an expert in that area in order to manage a large distributed system. This section presents related research work classified into distinct sub-areas of research corresponding to different aspects of the research performed in this thesis. Before discussing the related research works, existing middleware technologies that help develop distributed systems are presented and their shortcoming in fulfilling the vision of autonomic computing is discussed.

Existing Middleware Technologies

There are a number of different kinds of middleware tools and systems developed in the past two decades. They vary in terms of the programming abstractions they provide and the kinds of heterogeneity they manage beyond network and hardware. One such middleware technology for object oriented programming is Distributed Object

Middleware or DOM. DOM is the middleware that supports the object oriented programming paradigm and objects located on different machines. Distributed object computing is a computing paradigm that allows objects to be distributed across a heterogeneous network, and allows each of the components to interoperate as a unified whole. Distributed objects are like jigsaw puzzle pieces, because they can combine with each other, act independently and are portable. Distributed objects can combine to form part of an application or the entire one. The reasons behind DOM's popularity are [86]:

- Provides abstraction beyond those of the message passing system.
- Provides the power of OOP: encapsulation, inheritance and polymorphism, available to the distributed application developer.
- Enables clients to program distributed applications much like stand alone applications without the need for hard-coding dependencies.
- Growing focus on integration rather than on programming from scratch. This has the benefit of having less development time and more test time for the application programmer. Consequently the development cost decreases and level of support increases.

There are several kinds of DOM available as follows:

- Sun's Java Remote Method Invocation (RMI) [94] enables developers to create distributed Java-to-Java applications, in which the methods of remote Java objects can be invoked from other JVMs, possibly on different hosts. RMI supports more sophisticated object interactions by using object serialization to marshal and unmarshal parameters, as well as whole objects. This flexibility is made possible by Java's virtual machine architecture and is greatly simplified by using a single language. The newest RMI version is RMI-IIOP [96], which uses CORBA’s IIOP as its transport protocol compared to RMI’s JRMP protocol. RMI-IIOP provides an alternative for complex CORBA [70] implementations.

- Microsoft's Distributed Component Object Model (DCOM) [64] enables software components to communicate over a network via remote component instantiation and method invocations. Unlike CORBA and Java RMI, which run on many operating systems, DCOM is implemented primarily on Windows platforms.
- Simple Object Access Protocol (SOAP) [91] is an emerging distributed object middleware technology based on a lightweight and simple XML-based protocol that allows applications to exchange structured and typed information on the Web. SOAP is designed to enable automated Web services based on a shared and open Web infrastructure. SOAP applications can be written in a wide range of programming languages, used in combination with a variety of Internet protocols and formats (such as HTTP, SMTP, and MIME), and can support a wide range of applications from messaging systems to Remote Procedure Call (RPC).
- The Object Management Group's (OMG) Common Object Request Broker Architecture (CORBA) [70], which is an open standard for DOM, that allows objects to interoperate across networks regardless of the language in which they were written or the platform on which they are deployed. Like RMI, CORBA also allows an application to call a remote object's method and it provides a mechanism to access remote data fields of that object. CORBA is intended to be a generic framework for building systems involving distributed objects. The framework is meant to be platform and language-independent, in the sense that client stub interfaces to the objects, and

the server implementations of these object interfaces, can be specified in any programming language. The stubs and skeletons for the objects must conform to the specifications of the CORBA standard in order for any CORBA client to access these CORBA objects. As for RPC, CORBA defines an Interface Definition Language (IDL) through which a programmer defines any remote objects. Typically, a separate pre-compiler translates this IDL source code into C++, Ada or another language, and programmers link the resulting code with their applications. In CORBA, objects can never really leave their implementation hosts; they can only roam the network in the virtual sense, sending stub references to themselves and to clients. Users do not have the option of offloading an object from one host to another as in RMI. CORBA failed to make a greater impact on application programmers because of the tremendous learning curve associated with it and because it sometimes makes simple tasks more complex to programmers compared to non-distributed computation. In 1998 the OMG adopted the Real-time CORBA (RTCORBA) [87] specification, which extends CORBA with features that allow real-time applications to reserve and manage CPU, memory, and networking resources.

All the above DOMs uses some form of broker based system, where new objects are registered with a broker and all objects are invoked by obtaining the object location from the broker.

Automatic Partitioning

The following systems automatically break up the structure of a program into separate entities and execute them in a distributed setting.

Pangaea [93] statically analyzes Java programs and distributes the objects automatically on a networked system using a middleware mechanism. It derives an object interaction graph from the Java source code, which is an approximation of the program's runtime structure, and distributes the objects using remote communication mechanisms in a way such that inter-processor communication cost is minimized. The programmer can also override object placement decisions taken automatically by Pangaea based on the derived object interaction graph to satisfy any specialized distribution requirement. Pangaea also monitors interaction between objects at run time and migrate objects to complement the initial placement and to take advantage of locality.

Coign [43] automatically partitions applications built from binary components without accessing the source code. Coign uses scenario-based profiling to profile communication among components and based on the collected profile information, Coign partitions and distributes components to minimize the communication cost for a given distributed environment. Its applicability is limited as Coign only supports applications consisting of binary components that conform to Microsoft's proprietary Component Object Model [64]. Furthermore, Coign can only handle situations with two-host, client-server applications.

Addistant [100] is a Java byte-code translator for the automatic distribution of legacy Java software. It takes Java software to be partitioned and uses a separate user specified placement policy to translate it into a distributed version. Addistant requires placement policies be specified at the class level, limiting the opportunity to exploit object-level concurrency.

J-Orchestra [100] is an automatic partitioning system for Java programs. J-Orchestra operates at byte-code level and rewrites application code and replaces local data exchange with remote communication (e.g., Java RMI, indirect pointers). It uses information from static analysis and execution profiling to make the partitioning decisions. It requires no annotative inputs from users other than the network locations of various hardware and software resources. J-Orchestra also operates at class level granularity but unlike Addistant it does not require the user to explicitly specify policy for every class, instead it includes automatic analysis that ensures the correctness of partitioning.

The main difference between the approach taken by the above systems and the advocated approach in this thesis is that in addition to automatic partitioning, this research also aims to utilize resources efficiently. During the initial placement of partitions, none of the above systems consider processor and network load as selection criteria and therefore may generate a potentially bad allocation strategy. Object migration is supported by Pangaea only, but usually triggered by the frequent access between different objects, again not because of a heavily loaded machine or communication channel. The second important difference is that the above systems assume a static and dedicated environment, whereas this research considers a dynamic computing environment where resources may be shared with other users. In this unpredictable and dynamic environment machines can come and go and where a large number of them are dedicated to certain users thereby offering only a portion of their CPU time to the other applications running on this environment. All the above systems require the user to have complete knowledge about the hardware and software resources in their static environment; in contrast the system presented in this thesis automatically manages the resources in the system and relieves the user of the associated complexities. Among the automatic partitioning systems, only J-Orchestra provides partial fault tolerance, whereas, self-healing is a major focus of this research.

Automatic Distribution

In recent years, automatic program distribution has gained interest within the Java community as a means to design distributed virtual machines. Although most of the automatic partitioning systems intend to automate the code distribution, the systems actually do not provide any underlying infrastructure support for the distribution itself. Those systems are more concerned with program partitioning and not with the distribution of the resultant partitions. Here two of the prominent systems that try to provide distribution aspects along with program partitioning are mentioned.

JavaParty [78] introduces a new class modifier into the Java language, and a modified Java compiler translates such classes into Java/RMI remote classes, while also adapting any calls from client code. JavaParty offers a higher degree of distribution transparency than standard Java/RMI, as a result, making a class remotely invokable (i.e. distributed) requires no effort within the source code, neither on the client side, nor on the server side. JavaParty's approach only provides for initial program distribution and limited load balancing at run time. However, there is no support for other aspects of distribution or system management.

The shortfalls of JavaParty inspired the development of an automated distribution system known as AdJava [33]. AdJava shows that higher performance can be gained by providing automation and transparency in distribution of user programs. AdJava provides increased performance and throughput without any user involvement and without any change to the underlying middleware technology. This work also shows that by relieving the application programmer from complex middleware interfaces the system is more accessible to users who do not have advanced knowledge in middleware programming systems. Although AdJava shares some of the same drawbacks as JavaParty, the research in this thesis leverages off the experience gained in the AdJava project.

Autonomic Systems

There is no full fledged autonomic system either in the business domain or in the research domain that the author is aware of [83]. Most of the autonomic systems so far are actually prototypes or provide a limited amount of required functionality [58, 106] of an autonomic system. The most important aspect that is missing in all these systems is that the authors do not actually describe how to write programs in such systems or how to utilize such a system in a simpler fashion. They either introduce new metaphors or provide a completely new approach to autonomic computing that adds additional complexity and a steep learning curve to the programmer. The goal of this research is to make the resultant system simple to use, by making the underlying autonomic framework transparent. None of the following systems match this goal.

The Unity system [14] provides a platform designed to help autonomic elements interact with each other and their environment. It uses goal-driven self-assembly to configure itself. However, the utility function it uses for self assembly assumes that one can quantify the utility of different choices. The Unity system does not address the question of how complex it is for application programmers to use this prototype. There is no discussion of programming in such an autonomic system. Along with providing a runtime environment for autonomic elements, the goal of this research is to provide the programmer with simple to use interfaces to program in such a system.

Autonomia [26] is a ‘proof-of-concept’ prototype software development environment that provides application developers with tools for specifying and implementing autonomic requirements in a distributed application. The goal of Autonomia is to automate the deployment of mobile agents that have self manageable attributes. Autonomia only addresses self configuration and self healing properties of autonomic systems. Users of Autonomia have to use a well defined library and a predefined Application Service Template to create their programs. Therefore, users are exposed to the underlying system and need to know specific interfaces explicitly to program in Autonomia.

AutoMate [76] is an execution environment for Grid-based autonomic applications. AutoMate develops an autonomic composition engine to calculate a composition plan of components based on dynamically defined objectives and constraints that describe how a given high-level task can be achieved by using available basic Grid services. AutoMate provides a set of tools within a programming framework such as, autonomic composition and coordination middleware. However, as with IBM’s toolkit (Please see Autonomic Programming Environment Section later on), AutoMate does not address the complexity of integrating autonomic functionality into applications and introduces so many new metaphors and paradigms that eventually make programming such an autonomic system more complex than its current counterparts.

QADPZ [19] provides an open source framework that allows the management and use of the computational power of idle computers in the network using autonomic principles. QADPZ is implemented in C++ and uses MPI as its communication protocol, which restricts this system to a certain class of architectures. It also deploys a master-slave pattern for task distribution, which actually does not follow the autonomic system architecture and it does not take any measure to overcome a single point of failure, e.g. the master node. The clients and the slaves (which do the actual work on behalf of the client) talk to each other by the use of a shared disk space, which is certainly a performance bottleneck and requires costly synchronization.

Autonomizing Existing Systems

There is little research being conducted to add autonomic functionalities into existing systems and typically, the options explored are not transparent to the user and, in fact, require the user to have an extended knowledge of the existing code.

Haydarlou *et al.* [40] present an approach and a conceptual architecture for fault diagnosis and self healing of interpreted object oriented application. Their approach is to equip current and legacy interpreted object oriented code with their proposed technique so that the application can heal itself and can also attempt to solve the root cause that initiated the fault. Although this work shares similar goals and employs a similar approach, the solution provided in this thesis is extensive and completely transparent to the user. Details of code injection methodologies and different issues that arise during such code injection are addressed in this thesis. The authors in [40] propose a conceptual architecture for fault diagnosis; however, they fail to elaborate how the learning algorithm actually learns new fault scenarios. The authors experiment with their proposed technique with an application, for which there is no need to save any state information for a restart after a fault. Although the authors introduced a unique approach, in the form that they implemented may not work for more complicated programs, having nested try-catch blocks, local variable interactions, conditional branches, etc. The authors also do not provide any code inflation or execution time information for their proposed approach.

The work by Schanne *et al.* [83] attempts to inject autonomic functionality to existing object oriented code. The authors used the standard proxy/wrapper architecture to inject additional functionality, namely self-updating, self-configuration and self-optimization capabilities. The authors used static reflection to determine the structure of a class and then modify the methods according to their needs. One of the major drawbacks of their approach is that the user needs to supply the pre-processor with some meta-information about the code, such as, pre- and post-conditions and invariants of methods. The assumption is that the user has access to the application source code and could provide such meta-information about the methods. This assumption is not realistic for existing application code, as exploring the meta-information for such code is nearly impossible. Users can not get this required meta-information by simply examining the byte-code; source code is necessary to derive the meta-information, which in real life, may not be available for an existing code. Furthermore, as the authors in [83] rewrite only a portion of the byte code of the original class (either through mutators or directly and via proxy classes), all calls to the original class must be synchronized to ensure the consistency between the proxy and the original class. This sort of synchronization is costly and time consuming for frequent method calls.

Abbas *et al.* [1] build an infrastructure that offers adaptation, evolution and autonomic management support to existing systems. Their technique is based on dynamically linked libraries and therefore restricts itself to GNU C library. Their approach of injecting autonomic properties into existing code for run time diagnosis purpose is performed by inserting dynamic linker hooks in the existing code, so that at run time, corresponding libraries are loaded for diagnosis purpose. Since their approach is tied to C, it can not be used in a platform agnostic environment.

Kinesthetic eXtreme [55] provides a completely new approach for making legacy system autonomic. They use DASADA standards [6] for probes and gauge techniques to monitor and adapt an existing system. The existing code is retrofitted with probes and all probes in the system send runtime information to a centralized array of gauges where the overall system status is determined from the gathered information and any feedback to the system is relayed back to the corresponding components through the embedded probes. According to the vision of autonomic system [42, 58], each of the sub-components in the system should be self-managed and therefore, as a whole, the system is self managed also. However, this approach centrally collects status information from the whole system and then tries to change the behavior of the whole system. Since probes from all the sub-components are providing data to the gauges constantly, there is significant communication traffic. Furthermore the system provides self-healing capabilities at the level of the whole system level and not at the single component level. The self-healing approach provided in this thesis is completely different from the approach presented by Kinesthetic eXtreme. Instead of putting the monitoring and decision making code outside of the existing code, the approach advocated in this thesis injects code segments into the existing code. This allows finer level granularity with respect to control and make the resultant components fully self-sustained in any environment in which it is running.

Autonomic Programming Environment

Autonomic programming environments provide programming tools and development environments for building autonomic systems. Most of these systems provide support for heterogeneous platforms and limited autonomic capabilities. Although this research shares some of the same goals of these systems, its goal is to provide a transparent and easy to use underlying autonomic framework along with an associated development environment. This thesis presents a runtime system that adds those missing and required autonomic capabilities as an add-on or plug-in to the application and which modifies the user code transparently by injecting appropriate hooks and wrappers automatically into existing code.

IBM's Autonomic Toolkit [22] provides tools and an API for the purpose of monitoring, analysis, planning, and executing autonomic applications. The toolkit hosts several class libraries, plug-ins and tools for the Eclipse development environment [28] as follows:

- *Common Base Events*: The toolkit defines a standard data format known as Common Base Events [22] based on XML that define a standard data format for communication purposes.
- *Generic Log Adapter*: This plug-in for autonomic computing converts existing log files to the Common Base Event format.

- *Log and Trace Analyzer*: This tool reads log files in the Common Base Event format, correlates the logs based on various criteria, and displays the correlated log records in a user friendly manner.
- *Resource Model Builder*: This tool generates data models of monitored resources using Common Information Model format.
- *Autonomic Management Engine*: The Autonomic Management Engine hosts deployed resources models [22] and then monitors events on those models.
- *Integrated Solutions Console*: This is a web based user interface built on IBM's WebSphere Application Server [45] for a centralized management of autonomic capabilities.

Although these tools provide some inter-dependent functionality for an autonomic environment, they however, do not address the complexity of integrating autonomic functionality into applications. These tools do not help programmers to design their autonomic applications or to implement the logic required by autonomic elements. The toolkit also provides limited forms of communication primitives required for a more general autonomic environment. Developers of autonomic systems [62] agree that this toolkit is difficult to integrate into various distributed environments and requires real effort on behalf of the application programmer to use its functionality. The approach in this thesis is very different and attempts to relieve the application programmers from the burden of such complex and ever changing metaphors. It is our belief that programming an autonomic system should be made easy and transparent to the user, otherwise the goal of autonomic computing is sacrificed.

Orso *et al.* [73] present a technique to dynamically update an executing Java application. Orso uses proxy classes to rewrite application code and allows substitution, addition and deletion of classes at run time. Their technique operates by first statically modifying the application code (by class renaming and code rewriting) to facilitate its dynamic updating and then performs the hot-swapping of classes at run-time, when a new version of a class is available. Although this technique provides a novel way of reconfiguring an existing distributed system, it does not follow the autonomic paradigm and it needs direct user intervention to perform the hot-swapping. It also does not address all the other autonomic computing issues. This thesis employs Orso's technique for self-configuration and also for dynamic composition of the different components of the autonomic element.

The work by Griffith *et al.* [39] presents a technique to add self-healing capabilities to the Common Language Runtime, explicitly for the .NET platform. They provide a framework that allows a repair engine to dynamically attach and detach to/from a managed application for self healing purposes. Since the .NET platform provides more control to a user to access the runtime program and environment parameters, this technique does not work for a JVM platform because of its more restricted access to runtime environment.

Autonomic Element Architecture

There have been several autonomic system projects that describe autonomic infrastructure and define the architectural aspects of an autonomic element. However these research projects address the issue of designing an autonomic element with different objectives than those presented in this thesis. The goal of this research is to make the autonomic element simple to program and use and which, in turn, is sufficiently universal that it can be used in most cases. From the perspective of implementing the fundamental autonomic element, most of these related works are instructive. However, they typically do not clarify the design of the autonomic element necessary to build an autonomic system.

The work by Jarrett *et al.* [49] has the same objective as this research. The authors describe an autonomic computing architecture and accompanying implementation infrastructure on top of the Cognitive Agent Architecture [18]. However, they did not provide an architectural implementation of the autonomic element itself.

Wang *et al.* [104] propose an autonomic element design based on the mind agent model. It is a promising and intuitive approach to meet some challenges in autonomic computing initiative; however the authors actually did not implement the design to observe its workings in a real environment.

Accord [61] provides some good ideas on how to design autonomic elements and identifies the issues that should be addressed. However, Accord does not provide any architectural design. Similar to Accord's work, AutoMate [76] presents a prototype autonomic system without defining the internal workings of the autonomic elements. Autonomia [26] proposes a design of autonomic elements using proprietary techniques that add additional complexity and a steep learning curve for the programmer. The IBM Autonomic Toolkit [22] provides tools and an API for monitoring, analysis, planning, and executing autonomic applications. Although these tools provide some inter-dependent functionality for an autonomic environment, they, do not address the development of autonomic elements.

Existing research on multi-agent systems is a rich source of good ideas for system-level architectures and software engineering practices [14, 51] for autonomic computing. Systems such as Unity [14] use multi-agent paradigms for developing autonomic systems. However, these approaches typically lack common agreement on the exact design of most agents. The autonomic computing community could certainly benefit from their experience.

Self-healing Transformation

Traditional software fault-tolerance in Java is being actively studied by a number of researchers [28, 53, 78, 81, 85, 101, 103] in different aspects of software development. However, these approaches either work towards traditional stop and go fault tolerance or have different goals to this work. Typically these approaches:

- Operate at the source code level [28].
- Program through pre-defined interfaces to utilize such approaches [53].
- Deal with runtime software component update [78, 85]. [84] provides a detailed review of such approaches.
- Employ serialization techniques, which places extra burden on the programmers by requiring them to implement the serialization and de-serialization methods [101].
- Use monitoring and profiling during run time [55, 81].

− Use a modified Java Virtual Machine for better control over the checkpointing and recovery process [54, 103], however it scarifies portability and interoperability for doing so.

− Use different programming languages [1, 39, 61].

Please see the “Autonomizing Existing Systems” Section in this chapter for more information on those works.

The technique presented in this thesis differs from the work described above and from traditional fault tolerance approaches in the way that it relieves the application programmer from the burden of complex programming interfaces and ever changing metaphors by abstracting all such transformations automatically. In addition, programmers do not have to learn any new programming interface to utilize the technique advocated in this thesis. The aim is that any autonomic and self-managed system should be easy for the users to program, operate, and maintain, otherwise the goal of autonomic computing is sacrificed. Relatively little research has been done to integrate autonomic functionalities into existing user code and, typically the options explored by others are not transparent to the user and, in fact, require the user to have an extensive knowledge of the source code.

Other Related Autonomic Systems

There is further research being undertaken in area related to autonomic computing both in academia and in industry. Table 4 and Table 5 list such systems respectively. Although the functionalities provided by these systems do not match this thesis’s stated goals, the experience gained by these approaches is utilized to design and develop the system discussed in this thesis.

Table 4. Academic Research Projects on Autonomic Computing.

System	Description
AntHill [4]	This is an adaptive multi-agent based peer-to-peer system. The decentralized control techniques could be used in the system discussed in this thesis.
eBiquity [27]	Offers different artificial intelligence interaction techniques between mobile and pervasive computing entities.
OceanStore [71]	Focuses on working on large scale persistent data stores.
Wildstorm *et al.* [107]	Provides methods to handle variable workloads by dynamically reallocating hardware resources between machines.

Table 5. Industry Research Projects on Autonomic Computing.

Company	Description
Microsoft [65]	The AutoAdmin system makes databases self-optimizing and self-administering.
IBM [44]	Several prototype software architecture for autonomic systems and self configurable and self manageable server systems.
Sun [97]	Assists large data stores through service provisioning and policy management.
HP [41]	Helps customers in building system by autonomizing the process of business, service and resources.

How This Work is Different?

This thesis explores a new software tool for automating and separating the distribution concerns in programming distributed systems. The outcomes of this research benefit several areas of research and practice. Although the primary concern of this research is autonomic computing, the ideas explored by this dissertation are applicable to other domains, such as software engineering (system design, reengineering and software architecture), programming languages (meta-programming and code transformations), distributed and parallel processing (automating distribution issues and static analysis) and artificial intelligence (machine learning and policy management).

As scientific groups look to explore larger and more complex systems, their needs for computing support increases. This research looks to address the shortfalls of current middleware technologies by ensuring efficient utilization of resources including memory and communication needs as well as the distribution of the application. This improved support will meet the performance demands of these scientific groups and make their goals more achievable. There are several grid and autonomic computing projects in existence; however most focus on providing specialist middleware support and few, if any, examine automated support for distribution with transparent support for autonomic primitives. Furthermore, most approaches do not provide support for dynamic environments consisting of a network of heterogeneous machines. This research leverages off these existing software solutions and provides the necessary layer to hide the complexities involved in automatic distribution and autonomic self-management.

This research is significant in that it differs from the majority of the current research by trying to hide, as much as possible, the underlying distribution mechanisms and associated issues from the programmer, and incorporating a dynamic environment that consists of heterogeneous computing resources. In this research, there is no assumption that each of the computation nodes is equivalent in terms of performance or capability. The nodes are available for general use and so there is a need to dynamically monitor the machine usages and migrate autonomic elements as needed. In the presented system, the existing user code is analyzed and corresponding tasks are identified for distribution and they are then transformed into autonomous elements that handle associated issues automatically. The intention is not the development of the optimal distribution strategy for each application, but rather an acceptable distribution in terms of total elapsed time. The autonomic elements implement a policy whereby the program aspect they are monitoring is maintained within a range of acceptable values.

CHAPTER 5

SYSTEM ARCHITECTURE

This chapter introduces a novel peer-to-peer distributed object management automation architecture. In this model, independent or communicating objects are treated as managed elements within the geographically distributed autonomic elements. The presented organization offers significant advantages over traditional client-server organization by allowing the incorporation of self-management properties into each of the distributed nodes and making each Autonomic Element (AE) in the distributed environment equivalent in terms of managerial capacity. The unification of the traditional client-server roles allows management functions to be distributed across different elements in the system, thereby providing autonomous behavior of the whole system.

Requirements

To deliver support for autonomic behavior in a distributed object system, it is identified that the architecture should satisfy the following basic requirements:

1. *Support for expressing relationships among distributed autonomic elements*. The proposed architecture denotes each of the distributed autonomic elements as an individual service provider for the individual managed element assigned to that autonomic element. Users do not have to assign each individual program component to a corresponding service provider. A utility driven algorithm [23] is devised to map such services to provider mapping that uses service level agreements with a service

provider (autonomic element) before assigning it with a specific program partition (managed element).

2. *Support for management event notification in the distributed environment.* During the lifetime of the system, events for system management and accounting are generated and propagated across the system. To distinguish it from inter-autonomic element messages and to safe-guard it from network packet lost, a failsafe approach has to be formulated to propagate such event notification across the distributed environment.
3. *Support for consistent representation of element properties across the system.* Although in other related research, such information is represented with flat structures in the form of attribute-value pair, we utilize the Autonomic Computing Policy Language (ACPL) [2] to represent properties across the system. ACPL is a hierarchical tree model based on XML tokens that provide extensibility in the form of tree branching. Since ACPL uses a text based representation, it is easy to manipulate, space saving and faster in transmitting over the network.
4. *Support for autonomic element discovery, deployment, configuration and access control*. To support such system management issues, short messages have to be transmitted over the network at regular intervals. The implementation normally piggy backs or chains such messages together to reduce network congestion.

However, satisfying these requirements is insufficient as long as basic operational units for the programming framework for an autonomic system remain undefined.

− *Managed element*: A managed element (See Chapter 2) in the system is the minimum entity of execution, which comprises of one or more inter-dependent objects. During the lifetime of the system, managed elements can be broken down to smaller portions (or managed elements) and can be migrated to other autonomic elements to maximize system performance.

- *Communication*: The underlying communication mechanism should conform to uniform syntax and semantics and should follow open source protocol to fulfill the basic requirements set by autonomic computing principles [42, 58]. The Web Service Definition Language (WSDL) [105] is an XML-based communication protocol for describing network services as a set of endpoints operating on messages containing either document-oriented or procedure-oriented information that conforms to the requirement of our autonomic framework and is a candidate for future improvement. In the current implementation, Java RMI-IIOP is utilized to perform all communication between entities. Since XML data atoms (Chapter 6) are encapsulated into RMI packets and transmitted over the network to a destination entity, it requires the development of a WSDL translator to extend the functionalities in the future.
- *Naming*: The scheme for naming individual managed elements and autonomic elements is implementation independent and platform agnostic. It is dynamically generated and the name reflects some property of the entity (for instance source domain or number of objects) that it is representing. Currently this research is using simple domain specific name conventions [29] to implement naming of entities across the system.

Although other issues (such as security, protection and element discovery) related to a self-managed distributed system need to be addressed, in the current implementation, we are not addressing those due to the scope of this work (See Chapter 1).

Service Architecture

In this section, the architecture of the system is presented. The design is driven by the desire to produce a transparent and easy to use system that can be self-managed. The architecture provides services (physical resources) to the submitted workload (user application), where services are defined as an engagement of resources (service providers) for a period of time according to a contractual relationship (Service Level Agreement or SLA) with a service requester (application user). Figure 8 shows the overall system architecture. The architecture is broken into two views: *structural* and *managerial*. In the structural view, the lowest layer consists of the actual physical resources that are providing the services. This layer is abstracted by a virtual resource layer to hide the interface complexity of the actual physical devices. Having this virtual

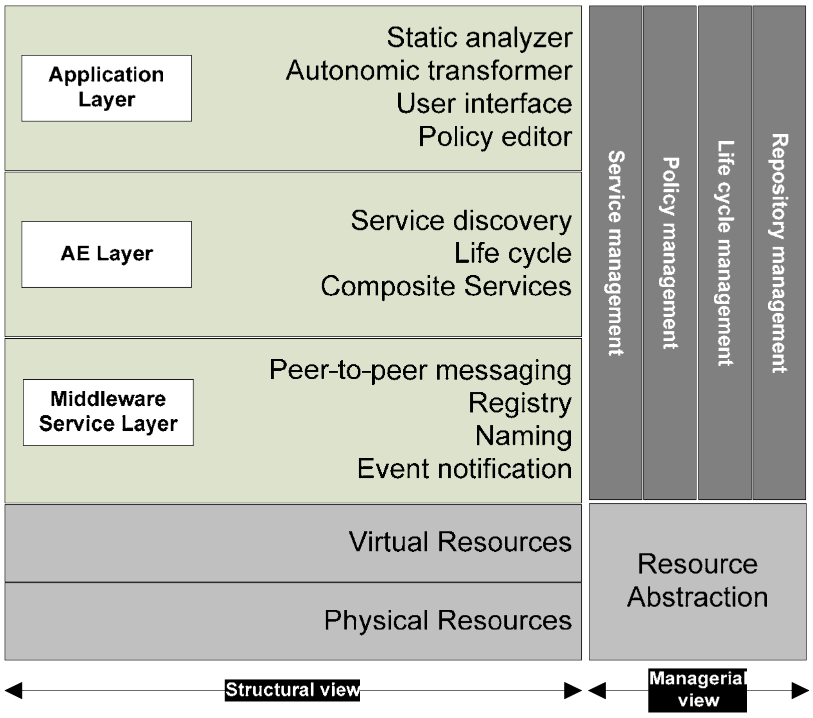


Figure 8. Autonomic Service Architecture.

layer with a pre-defined interface allows the architecture to incorporate new physical devices in the future without re-programming the whole architecture for each new device. The upper layers provide a set of predefined services dependant on the state the current autonomic element is currently in. Some of the autonomic elements are directed to act as managerial elements to provide certain services across the entire system, so that other autonomic elements can work smoothly and can provide the service as requested. The application layer, normally kept dormant, can be brought alive if needed by either the user or by another autonomic element. The autonomic element layer is responsible for the management of autonomic elements and general system wide management. The middleware service layer performs actual communication services which are used for inter autonomic element communication, repository update, notification of migration of managed elements etc. The managerial view consists of the management functions necessary to manage the services delivered. Management functions primarily involve policy management, resource management, service management and life-cycle management. All management functions are performed by autonomic elements which are self-managing and whose role is to ensure automated delivery of services.

Figure 9 shows the hierarchical management view of the architecture. The global autonomic manager acts as the interface between the user and the underlying system. Each of the program partitions is treated as a service instance and encapsulated by an autonomic element for self-management. Each of the service instances are assigned to resource domains that best suit the instance's needs. See reference [23] for more information on devising such service instance to domain composition details. Each of the

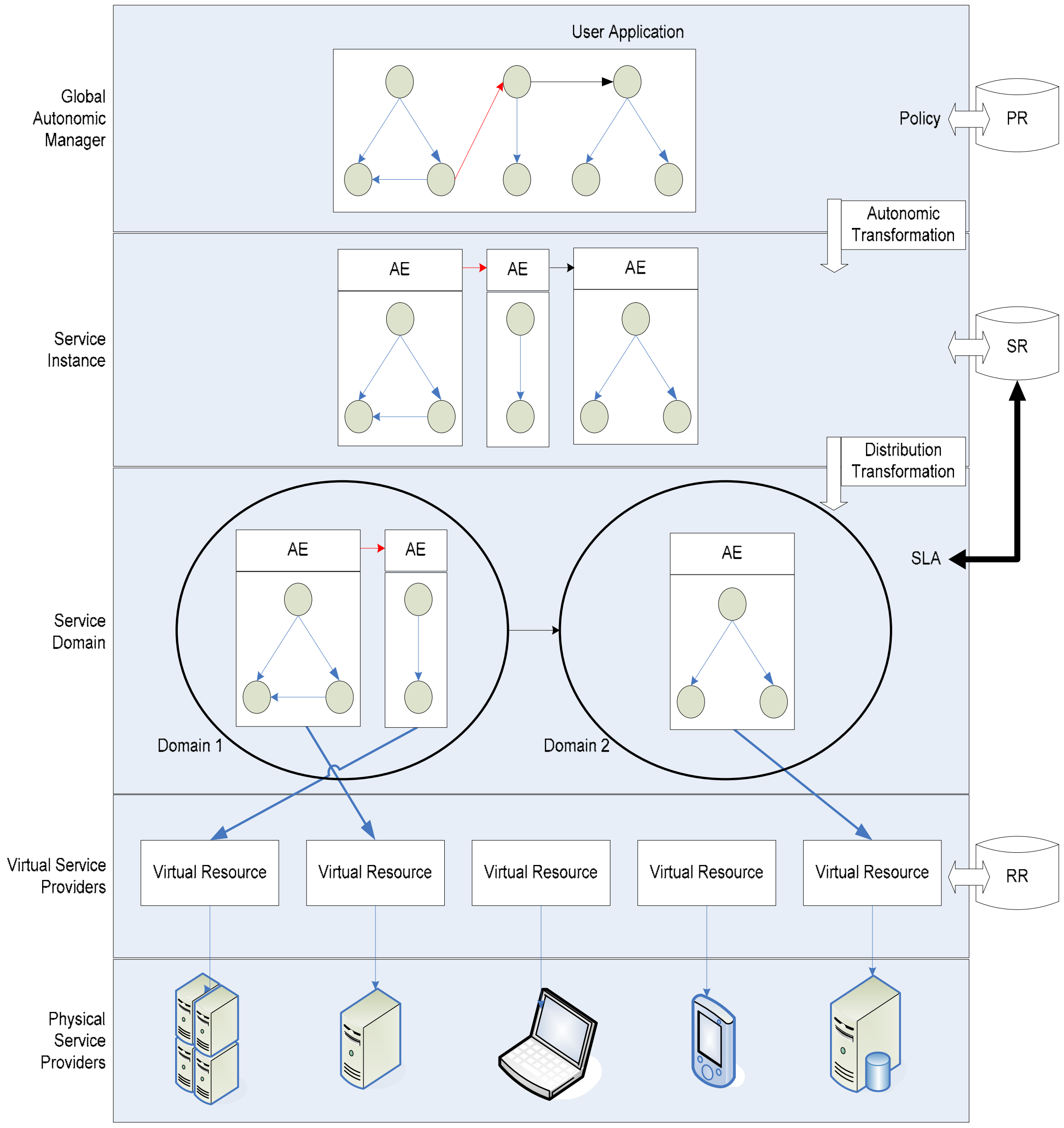


Figure 9. Hierarchical Management of System.

actual physical resources has a corresponding virtual resource adapter that provides a uniform and consistent interface to the physical resources. This simplifies resource management, service composition and dynamic resource addition/deletion. In order to perform autonomic service management, the system must maintain appropriate information about the different components of the runtime environment in repositories (databases). Repositories are classified into three distinct types:

- *Policy repository (PR)*: This repository contains the policies created at runtime or pre-defined by operators or users following a template policy implementation. The user can browse through existing policies and can use them to create new policies using a policy editor interface.
- *Service repository (SR)*: The service repository contains information (such as parties involved in a service level agreement) about the different service instances activated by the global autonomic manager, types of resource used, amount of resource being used and past operation history.
- *Resource repository (RR)*: This repository holds information about the resources available to the service providers at any given time. Information that may be stored for each resource includes type of resource, resource performance matrices, communication matrices, etc.

The implementation of these repositories is performed hierarchically to avoid a single point of failure. Usually each service domain has its own copy of the repository and inter-domain repository information is passed only when the system is idle or there is a need for service re-distribution.

System Operation

After the user presented the code to the system (at the global autonomic manager), a static code analyzer builds an object graph from the user supplied byte code. See reference [24] for more details on the static code analyzer. Once it generates the object graph, the graph is partitioned according to the underlying system configuration, communication requirements or any user supplied policy. The underlying system comprises a collection of platform-agnostic autonomic elements as an interface to the service provider and the associated pre-processor for comprehensive byte code to byte code translation, so that the resultant transformation produces a self-adaptive version of the user code. The transformed program is based on self-contained concurrent objects communicating through standard object based communication protocols and incorporates salient features (such as Broker architecture and asynchronous call) from existing middleware technologies. Then the global autonomic manager communicates with the underlying system for various runtime parameters (such as the number of service providers, service provider information and domain information) and generates an initial deployment scheme detailing aspects such as object placement, selection of service provider, target node and communication protocol. Once such a deployment scheme is generated, the underlying autonomic framework performs automatic application partitioning and placement based on site-specific application placement policies, capabilities, and current system load. Once the program partitions are distributed, the underlying autonomic system gains control of the objects and manage the distributed program thereafter. As a result the underlying framework is adaptive, since it adapts the user application to various platforms and protocols on the network and to unpredictable runtime conditions (See Chapter 7). Figure 10 shows the system's flow of operation. After the static analyzer has created the object graph of the user program, the autonomic transformer transforms the corresponding partitions into managed elements assigned to a specific autonomic element and directs it to execute its partition with a given policy.

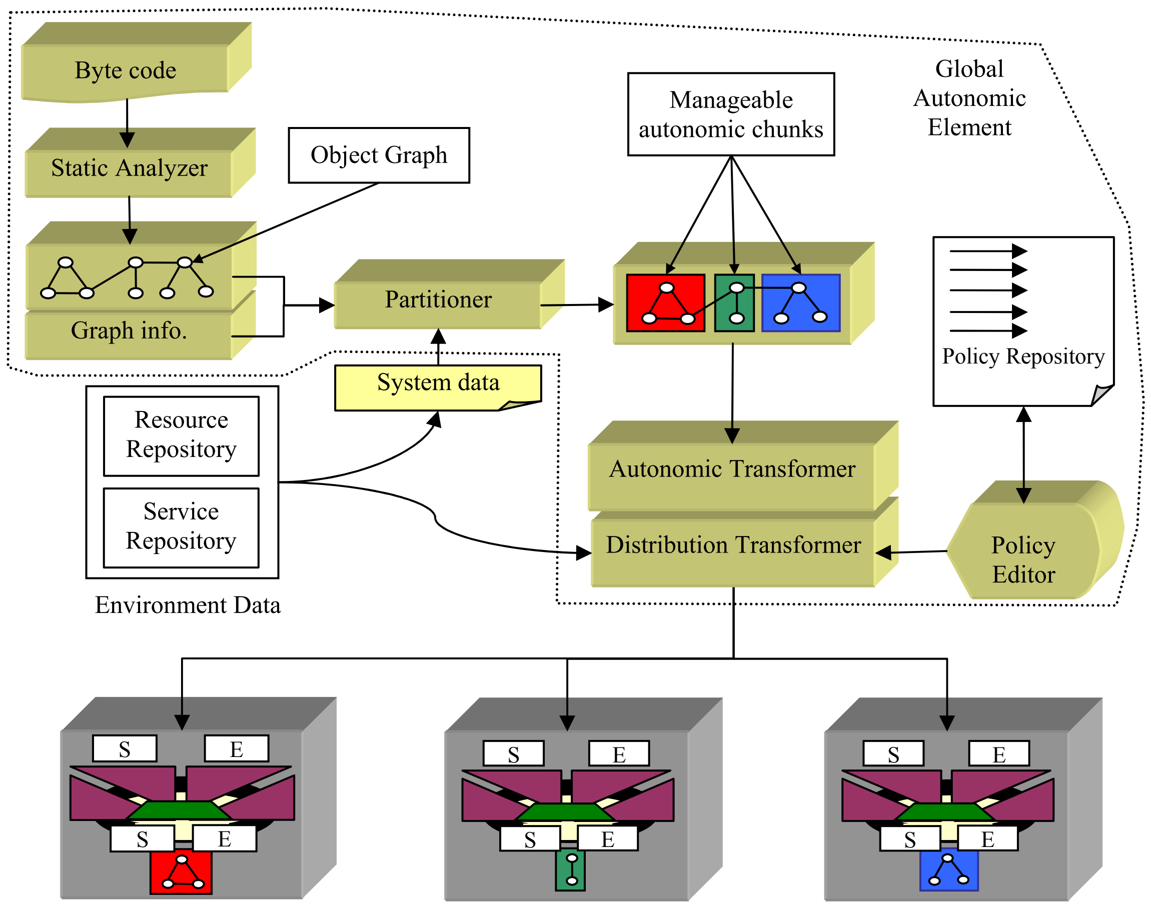


Figure 10. Flow of Operation.

Policies

An important aspect of any autonomic system is for the user to specify the behavior of the system at a high level with broadly scoped directives. The benefit of policy-based management is that the behavior of the computing resources can be guided to follow certain rules, and dynamically configured so that the system can achieve specific goals and can react more promptly to environment changes. The system allows two forms of policies to cover most of the possible contexts. The first form of policy is the *if-then-else* policy and is static in nature. These policies are static and have nearly no abstraction and describes actions based on the value of a predicate. Examples include: distribution decisions dependent on a certain number of service providers in the system; what to do in case a service provider joins or leaves a domain; a domain becomes unreachable. The second form of policy is more abstract and can be composed of several if-then-else policies. This form of policy can be changed during runtime depending on service demand change and varying loads. Chapter 7 provides an example of such a policy. The Autonomic Computing Policy Language (ACPL) [2] is adapted for policy management as it provides a user friendly form of a policy definition, policy management and different tools as well as an API to work with policies. Currently the user of the system can define their own policies through pre-defined interfaces and templates. However, future development of a graphical policy editor can overcome the lack of a simple interface for the users of the system. The policy manager requires the following components along with a policy editor for better management of system policies:

- *Policy validator*: The role of this component is to ensure that no conflicts arise due to redundant or duplicate policies or inconsistency in policy definition.
- *Policy adaptor*: The role of this component is to translate policies defined in other standards to the adopted standard or export existing policies to other standards for cross domain deployment.

− *Policy distributor*: This component is responsible for proper delivery of policies across the system and accumulation of policies from across the system and saving them consistently in the policy depository.

Currently, the static analyzer and the partitioner provide some basic policies, comprising of resource allocation, computation and communication directives derived from the analyzed code. Users can add or modify these system generated policies and can supervise the deployment. Incorporating different machine learning techniques will assist the system learn new policies over its life time.

Management Operations

In this three tier peer to peer architecture (Figure 8), each autonomic element can also act as a managerial element. All autonomic elements operate over a unified management model that provides a set of operations and interface common to all autonomic elements. Normal autonomic elements access this model to retrieve their current configuration and save their current state, whereas managerial elements use it to discover other elements in the system. Representation of management information is a crucial part of any self-management architecture. The representation should be easy to manipulate and should follow an open standard for faster incorporation and future extensibility. XML-based WSDL is used to represent such management operations for these reasons. Currently, there are four types of management information kept by the system:

1. *Resource information*: Resource and service provider properties are expressed by this type of information. Care should be taken so that there are no duplicate

entries and periodically (heuristically decided) this information needs to be updated to keep it consistent and non-redundant.

2. *Performance information*: This type of information represents the performance status of a running autonomic element. Before two autonomic elements come to a service agreement, they transfer performance information to learn more about each other, which includes different performance measurements and current operational state.
3. *Configuration information*: This type of information is used to configure the behavior of an autonomic element. Concurrency control has to be in place so that multiple configuration information is in play at the same time.
4. *Relationship information*: This type of information expresses dependency relationships between autonomic elements.

A model similar to DNS revolver is employed to discover autonomic elements in the system. Each domain has its own name server and name server in different domains correspond to each other to solve inter-domain name discovery. Nodes in the same domain are listed in the name server of that domain and very node in a domain is kept aware of the designated name server for that particular domain. This is the most reliable way of discovering elements in a domain centric environment.

Autonomic Properties

The four main properties of this autonomic system are discussed below.

Self-configuration

To use the autonomic resources, potential users must first register their computer with the autonomic system through a user web portal. This makes their computational resources available to other users while granting them access to other machines. The intent is that the computational resources will only be made available to other users while the computer is idle (amount of CPU utilization is less than a certain threshold). There should be no observable degradation of performance apparent to the user of the computer. A user may deregister their machine at any time and consequently the autonomic element running on that machine will delegate its current managed element to other available autonomic elements without the loss of useful computation. Figure 11 shows the scenario of a user registration into the system. For a first time user of the system, the process of registration into the system is as follows:

1. A web portal for the system is maintained. Users visit the portal and register that particular resource into the system.
2. A unique identifier for that particular resource is created and added to the resource repository. It is verified that a single resource does not have more than one entry in the resource repository.
3. The user is notified of the registration status and given a user name and password combination (authentication information) to utilize the system in the future. An autonomic element in the form of an executable program is then downloaded for that particular platform and installed by the user. Once installed, the

authentication information is used by the autonomic element to join the distributed system.

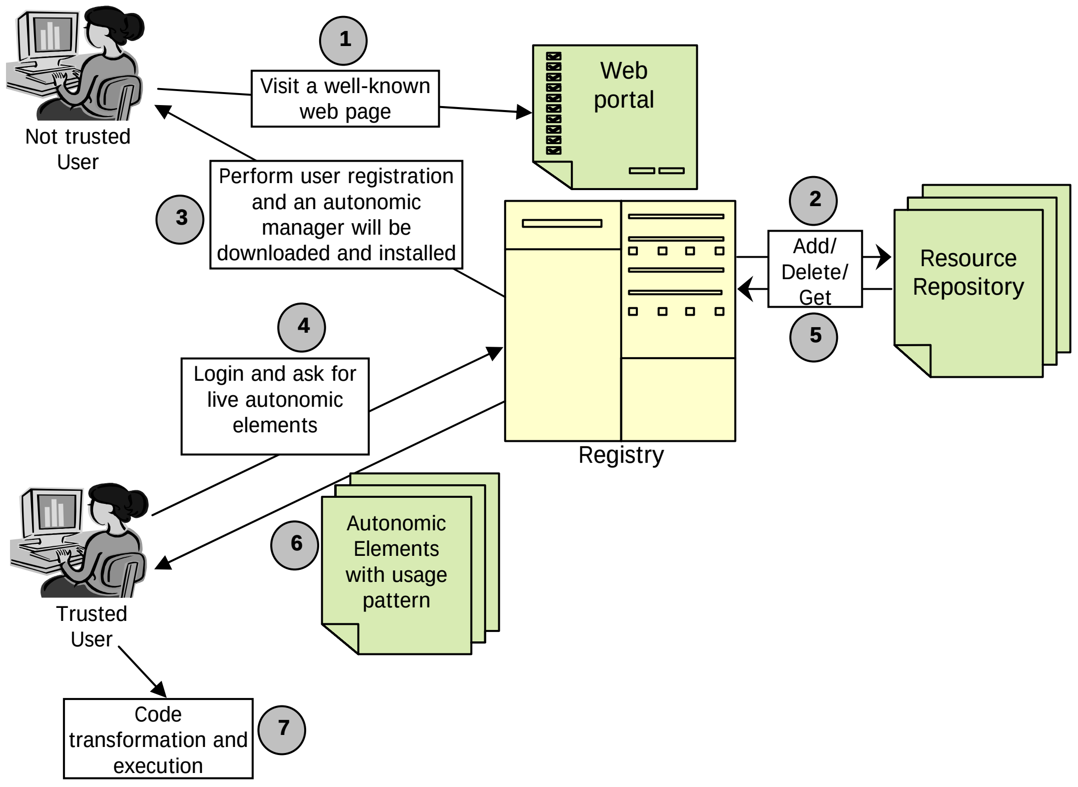

1
Visit a well-known web page
Web portal
Not trusted User
Perform user registration and an autonomic manager will be downloaded and installed
3
2
Add/ Delete/ Get
5
Resource Repository
4
Login and ask for live autonomic elements
Registry
Autonomic Elements with usage pattern
6
Trusted User
Code transformation and execution
7

Figure 11. System Startup Process.

Once joined to the system, the process of submitting a computational task to the system is as follows:

4. Assuming the user is logged into the system, the user application is submitted for conversion and execution.
5. A list of available resources is requested from the resource repository.
6. The resource repository sends a list of resources with the usage pattern and contact information for each resource.

7. After that, the user code is delivered to the static analyzer and follow up code transformations and program executions proceed as usual.

Dynamic allocation of a partition (managed element) to an autonomic element is required since the environment is one that is shared by other users. Flowchart 3 (Appendix A) shows the dynamic process of setting up a managed element within an autonomic element. Consequently, the autonomic element load may vary over time and some autonomic elements may be unavailable at the time of execution. The autonomic element characteristics and load patterns are learned and retained in a database which is used to determine potential candidates for the distribution of a program partition based on the resources it requests. Consideration of specific conditions is taken into account before allocating a partition to an autonomic element. These conditions include factors such as the need for particular objects to be co-located within a single autonomic element, the need for specific hardware architectures (e. g. shared memory) or the need to access large data repositories such as satellite information that can not efficiently be shipped to the program partition.

Self-healing

The goal of self-healing is to produce resilient autonomic elements and a resilient autonomic system as a whole; however there could be situations, from time to time, when it may fail (like electricity outage). However, the system does not provide any single point of failure as it replicates all the essential services and saves those replicated copies in a hierarchically implemented repository. The challenge is to make the system work as a whole, although part of the system may have failed for any unforeseen reason. There are numerous fault scenarios [88] that the self-healing property of any autonomic system has to cater for. This thesis focuses on proof of concept by providing support for a small number of predetermined faults that are transient in nature [111]. Therefore, the system in its current implementation is covering the following fault scenarios:

1. The user code produces some runtime exceptions. Chapter 7 extensively describes corresponding code transformations and self-healing approaches.
2. The autonomic element crashes. At regular intervals, each autonomic element status is saved so that after a crash, it can restart from the last saved state.
3. The resource, on which an autonomic element is running, crashes. The same policy as above is used for this scenario also.
4. Different repositories are unreachable. Since all repositories are implemented hierarchically (each domain has its own repository and repositories in different domain share information only when needed), the repository on top of the current domain is accessed in such situation. However, if ambiguities arise for such access, or repositories in other domains are also inaccessible, then a random amount of time is waited to check whether repositories are accessible again.

Self-optimization

The system provides self-optimization at the autonomic element level by being light-weight in terms of resource requirements and continuously monitoring itself and dynamically changing its mode to reflect the monitored status. It is recognized that good optimization behavior by each element does not necessarily guarantee that the whole system will be optimized. For system wide self optimization, the system would also need to resolve any conflicts that arise between two autonomic elements. To make the scheduling process dynamic, periodic exchange of state information among the machines is necessary. However in doing so it is necessary to take care not to overwhelm the transmission medium to the extent that the overall system suffers performance degradation. Although broadcasting state information is the most widely used approach, it tends to cause congestion and it does not scale well. There is no need for an autonomic element to exchange its state information with all other autonomic elements in the system. Rather, each machine's usage pattern and load characteristics are monitored over time and a general pattern is deducted by each autonomic element. This information is saved in the resource repository along with additional machine information (such as machine characteristics, operating system installed and JVM type and version) and shared among autonomic elements. Therefore, when an autonomic element needs the cooperation of another autonomic element for load balancing or other optimization, it chooses one with a usage pattern that satisfy its needs. Once it finds a desired autonomic element, the final service relationship is developed by polling each other with pre-determined service primitives. This reduces network traffic and allows autonomic elements to response quickly to changes. The static analyzer [24] employs a utility driven algorithm [23] to optimize initial program partition placement. The same algorithm is used to place sub-partitions in other autonomic elements if later, during execution, it is necessary to break up larger partitions into smaller ones to optimize program execution.

Self-protection

Security is a major concern for any autonomic distributed system. Security is necessary to guard against malicious attack and faults. Security is needed on two fronts: the autonomic element itself and the distributed infrastructure. All the autonomic elements require access control and authentication during communication. Since autonomic elements communicate by building a relationship with each other; the nature of the relation (*initiate*, *setup*, *status*, *terminate* etc.) provides a set of access control primitives. Since each of the service relationships has a pre-defined set of communication primitives, malicious attacks (which are not part of the per-defined set) are treated as unrecognized commands. Similarly, the resources over which the application is distributed may be subject to attack by intruders and it is necessary to ensure their integrity also. In order for such distributed systems to be effective, the issue of security, or more specifically integrity, confidentiality, and availability, must be addressed. The issue of availability is addressed by the fault tolerant nature of the architecture. However, controls at various levels, such as the operating system, application, and network are necessary to guard against integrity and confidentiality failures. The thesis does not address self-protection at this moment due to its overwhelming research scope.

Distribution Transformation

Once a group of objects has been identified as a managed element of an autonomic element, the corresponding class files must be transformed accordingly to facilitate insertion of autonomic primitives into the code. The static portion of the class is first moved to a separate class by the static analyser [24] and corresponding redirection is established to preserve the semantic and functional flow of the original execution. Communication between distributed objects is permitted, but such interaction should be kept to a minimum in order to reduce the communication overheads. The assumption is that the distributed classes are multi-threaded to allow asynchronous execution. This assumption is true for the problem domain of the research where the whole program is modelled as a collection of concurrent objects,

For each of the distributed objects, the code transformer transforms the byte code by instrumenting code to provide autonomic properties. Specifically the points where distributed objects are declared and called are now transformed to call and invoke corresponding proxy object classes. The proxy can itself redirect that call to a distributed autonomic element depending on the system level distribution policy. Every distributable class is transformed and the distribution transformer makes the following changes:

- The distributed class is modified to implement a new interface which holds distribution primitives.
- Any input/output statements in the user code are marked for redirection as specified by system level policy. The reason for redirecting all input/output is explained in the next section.
- The constructor is replaced with an *initialize* method.
- Based on the parameter-passing mode (*pass-by-value* or *pass-by-reference*), all parameters and corresponding local variables are changed. Since RMI uses object serialization to pass parameters and return values in RMI calls, it can pass the true type of an object from one virtual machine to another in a remote call. In RMI, a

non-remote object that is passed as a parameter to a remote method invocation, or returned as a result of a remote method invocation is passed by copy. So, in this case the content of the non-remote object is copied before invoking the call on the remote object. When a non-remote object is returned from a remote method invocation, a new object is created in the calling virtual machine. On the other hand, when passing a remote object as a parameter or return value in a method call, it is passed by reference. However, there are situations when non-remote objects are required to be passed by reference. For instance, if a non-remote object is passed as a parameter that itself refers to other non-remote objects, then the whole graph of objects is copied to the recipient. If the receiving method changes the graphs, then there will be different versions of it in the system. As RMI does not support pass by reference for non-remote objects, the code transformer employs RMI's callback feature [94] to simulate pass by reference for non-remote objects.

– All public methods in the distributed class are modified to throw a *RemoteException* (to satisfy RMI requirements) and *MigratedException* (to satisfy runtime migration) along with any other existing exception thrown by the method.

– To make remote objects migratable, they are transformed to implement the *Serializable* interface. The code transformer checks whether the class itself or any of the class in its inheritance tree implements *Serializable*, and adds it if necessary. However, not every class in Java can be made *Serializable* in that way because of the language level constraints. Such objects have to be anchored in one single node during the execution of the program.

Other than making these changes, the remainder of the code remains intact. As a requirement of RMI, all methods must be public so that they can be called remotely. So the code transformer checks that all access rights are maintained. Only after the semantic check and transformation, the methods in the generated class are made public.

Autonomic Transformation

The static analyzer converts computationally intensive, large parallel applications and models them as a collection of independent computing and communication resources with diverse capabilities within a large-scale integrated system. Since the transformed program runs as a distributed program, any single node in the system could have one or more objects executing on it. As all of the related objects within any single node have to be manipulated transparently, several different scenarios present themselves for consideration:

1. *One AE per node with each object as a ME (Managed Element)*: The traditional proxy approach [102] could be employed; however, the full computation power of a particular node is not utilized in this scenario. The traditional proxy approach only supports a single object per proxy, but multiple objects per proxy are required for the approach presented in this thesis.
2. *One AE per node with multiple objects as ME*: This is the most common scenario where multiple objects run concurrently in one node. The traditional proxy notation must be extended to address this scenario.

3. *Multiple AEs per node with multiple objects as a ME*: Although this scenario is possible, it is not supported by the autonomic computing paradigm and is not considered further.

A proxy structure is implemented that encapsulates one or more runtime objects into a single manageable entity that communicates with other such entities in the system with a single communication channel provided by the proxy object. Having an encapsulating proxy object allows us to incorporate the autonomic functionalities seamlessly into the user objects with the help of sensors, actuators and control interfaces. Once user classes are transformed using the traditional proxy approach, a new proxy class is created to encapsulate all the other proxies in that node. Any inter-object communication inside a single ME proceeds as usual; however, any inter-object communication between two different MEs is delegated to the encapsulating proxy object. There are two possible choices to create such an encapsulating proxy:

1. *The proxy class inherits the original class*: This works for only one class as Java does not support multiple inheritance. This can be overcome by creating a new interface with all the methods of each of the target classes and then having the proxy implement that interface by copying the method's body into the proxy class.
2. *Renaming the original class with the proxy class*: This is the traditional approach and does not work without modification for the proposed approach as it is difficult to delegate all proxy invocations separately. This approach needs to be extended by creating a clone of the original class structure as the new proxy structure and, at the byte code level, redirect all calls to the proxy class. This

allows the existing methods to be overridden with additional functionalities and the class to be extended with new methods. Since all associated transformations are performed at the byte code level, users do not need to be concerned about following any specific programming rules.

For such transformations of user code, the following issues need to be addressed:

1. *Methods (M) and constructors (C)*: There are more methods in the proxy class to interact with the AE and also to manipulate the object itself. So,

   if M_n , $C_n \in$ *Original Class* $\rightarrow$ $M_{n'}$, $C_{n'} \in$ *Proxy Class,* where $n' > n$.

   The original methods are overridden with the following structure:

   Pre-processing

   Original method call

   Post-processing

   Instead of instrumenting each method, a wrapper method is created to ensure consistency with the existing line number table for debugging purpose. Figure 12 shows a possible proxy creation scenario. The original object graph (Figure 12 (a)) is transformed to a corresponding proxy object relationship. More information about generating object graphs and how to partition such graphs can be found in reference [24]. Next, each partition of the graph is encapsulated with a master proxy (Figure 12 (b)) to add extra functionality and to have a second level of delegation. So, if there was a method call in *U1* such as *X2.foo(arg1,arg2)*, it is transformed to *Proxy_AE1.invoke(X2, foo, args)*, where *args* is an Object array having all the arguments of the calling method wrapped appropriately with their object classes. The invoke method inside *Proxy_AE1* identifies which node *X2* is located in (node 2 for this example) and makes a remote call to the method *Proxy_AE2.invoke(X2, foo, args)*. The remote invoke method unwraps the arguments and calls the local method *foo* in *X2* and wraps any return value from that method and sends it to the caller (*Proxy_AE1*). Having an encapsulating proxy allows migration of an object from one node to another (Figure 12 (c)) for optimization purposes by creating a new proxy wrapper in the new node that encapsulates the migrated object. This permits invocation of a method even if the original object has migrated to some other node and the original proxy is left somewhere else.

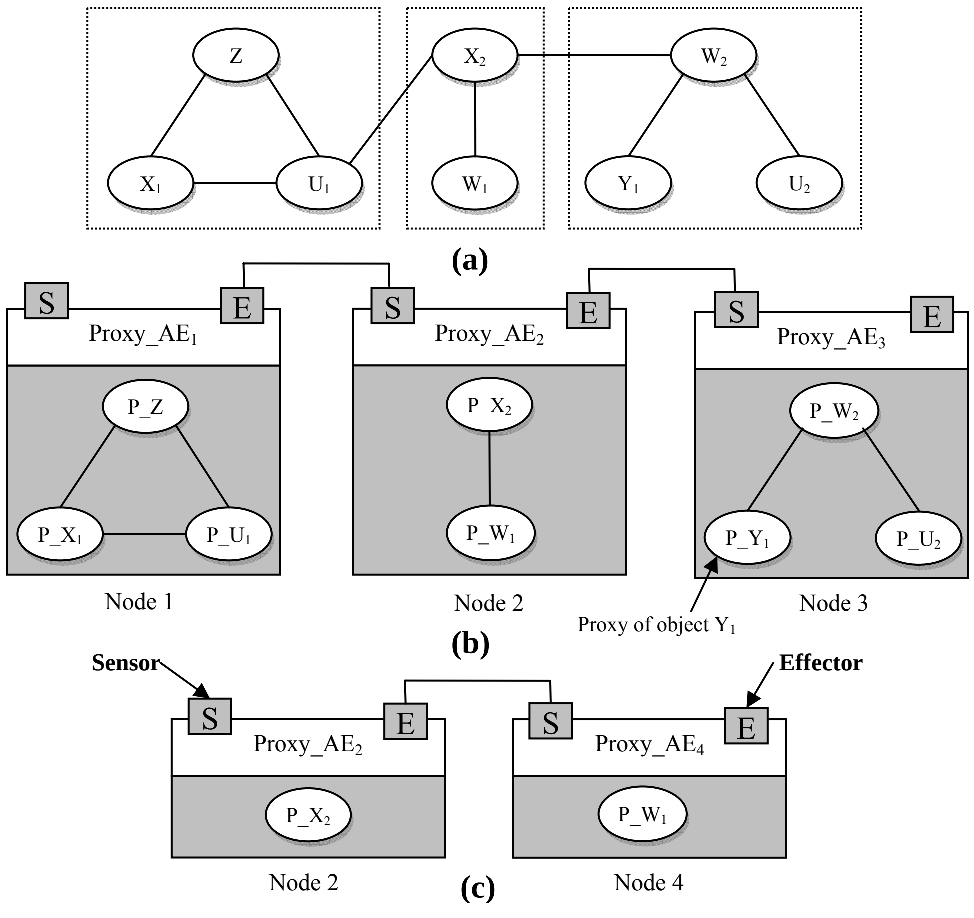


Figure 12. Object Transformation.

2. *Polymorphic method calls*: To determine the original calls of a method in a polymorphic call requires a stack oriented emulator such as that in the JVM. Creating such an emulator is a separate research problem and in the initial version of the transformations, polymorphic method calls are not considered.
3. *Direct field access*: All direct field access is converted to getter and setter methods to facilitate remote method invocation.
4. *System classes*: Since the system classes cannot be modified, the same techniques as used in J-Orchestra [102] are adopted. System objects are either moved with the user objects or, if they use any platform dependent code, remain on the same node and other proxies access these system objects using a callback facility.
5. *Handling distributed I/O*: It is undesirable to have user code produce output in a remote machine or ask for input somewhere other than it is intended to. Therefore all input/output operations need to be redirected. This leads to the following possible transformations:
    a. *Standard output and error*: The user has the choice of redirecting all remote output/error to the client machine, saving it in a log file in the remote machine or ignoring it as a whole. Redirection to the client machine is undesirable as this may overwhelm the network with excess traffic.
    b. *Standard input*: All standard input is redirected to the client machine.
    c. *File input*: The user has the choice of redirecting all file read operations to the client machine or has the underlying framework copy the file to the remote machine and have file input operations precede as usual. Both of

these choices have their own tradeoff and one is better than the other in different situations.

d. *File output*: File output could be redirected to the client machine or saved as a local file in the remote machine and moved to the client machine at the end of the computation.

6. *Exception handling*: Chapter 7 illustrates the approach adapted for handling exceptions. Users could utilize different system level policies to steer the exception handling approach, such as permit the exception to propagate through the class hierarchy or try to heal the exception with user interaction or previously saved solutions.
7. *'final' class*: To extend the functionality of a un-modifiable class in the proxy, the final keyword is removed from the classpool [50] inside the class file. In this way, the semantic and functional consistency of the class remains the same but now extra methods can now be added and existing methods can be overridden in the proxy to add the extra functionality.
8. *Existing interfaces*: Since Java allows a single class to implement as many interfaces as required, no changes are required in this case.
9. *Static methods and fields*: Any class that has static methods and fields is divided into two subclasses where one has all the static methods and fields and another has the non-static entities. Separate proxies are created for each subclass and the static subclass is anchored in one node and interacts with other proxies using RMI callbacks.

10. *Use of 'this' and 'super'*: Use of *super* does not cause any problem in the transformed code. However, the use of *this* needs to be delegated to the appropriate proxy class.
11. *Use of reflection*: Reflection in the user code is not considered due to the added complexity in the byte code rewriting phase. A separate package on reflection that delegates user enforced reflection must be developed to address this issue.

To handle other Java language features, the techniques used have similarities with the techniques used in J-Orchestra [102]. The major distinction with the approach in this thesis is that object level distribution is attempted, whereas J-Orchestra uses class-level distribution. One significant drawback with both of these approaches is that it incurs extra space and time cost to run the resulting distributed application. Since new proxy classes are created and segments of byte code are inserted into the exiting byte code, some inflation in the resulting code size is expected.

Performance Analysis

Experiments were conducted to find the time requirement of delegated method calls using proxy-based managed elements. After code transformation, every remote method call now takes place through the autonomic element. Therefore the call can be broken down into several parts:

1. Call to local proxy.
2. Call from local proxy to remote proxy (AE).
3. AE locates the remote class, loads it and uses runtime reflection to delegate the execution to the corresponding method.

To explain the process, consider the UML diagram of an example program presented in Figure 13 (a). The application class (which is the starting point of this application) creates an instance of *classA* which is run as a threaded object. For instance, we like to transform

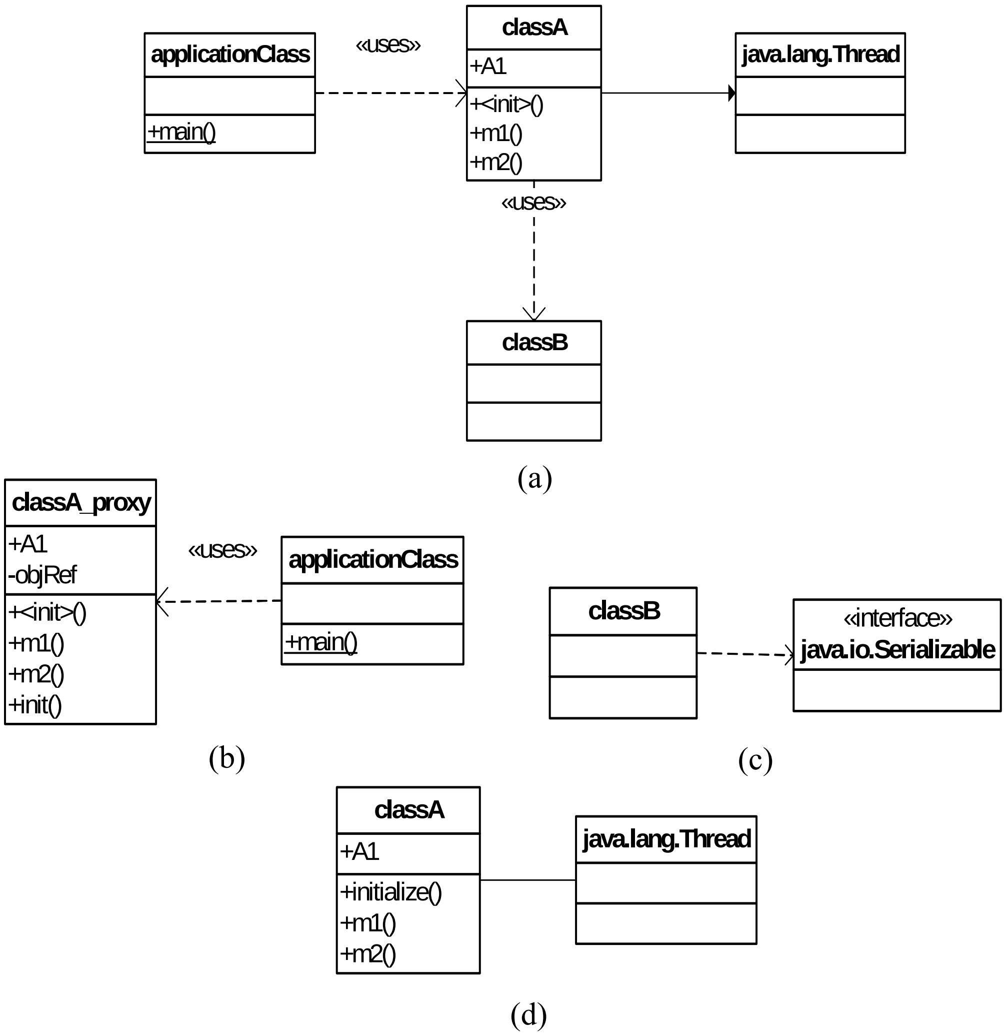


Figure 13. Example Class Diagram.

transform the instance of *classA* into a managed element of an autonomic element. The class diagram is now transformed into three separate portions with separate transformations as follows:

1. *Application Class*: Instead of referring to *classA*, it will now refer to the proxy instance of *classA* (Figure 13 (b)). A new wrapper class is created for *classA* that acts as the proxy of the original class. Along with the traditional proxy transformation, a new private variable that holds the reference (local or remote) of the actual object, and a new method that helps initialize the actual object reference, is inserted into the proxy class.
2. *Used classes inside the managed instance*: Objects of any class that *classA* is using must now implement the *java.lang.Serializable* interface to help the system to transfer the instance over the network if needed (Figure 13 (d). All such classes are checked to see whether they implement either the *Serializable* interface or a subclass of that interface. If not, then it is added automatically. If such classes use any object instance which is not serializable (for instance having another thread inside that object), then that particular managed element must be anchored on a single machine during the life time of the application.
3. *Original class*: The original class relationship is kept unchanged (Figure 13 (c)). Only the constructor is now replaced by a new method *initialize* to create an instance of the class during runtime by the autonomic element using a consistant interface.

A new class is designed to house the managed object instance which implements a pre-defined interface. The *AutonomicElement* class creates an instance of the *managedElement* class. Figure 14 shows the runtime structure of each proxy class in the system. The *init* method in the proxy class is responsible for gathering necessary information (such as location of an available AE and naming of the AE) and establishing

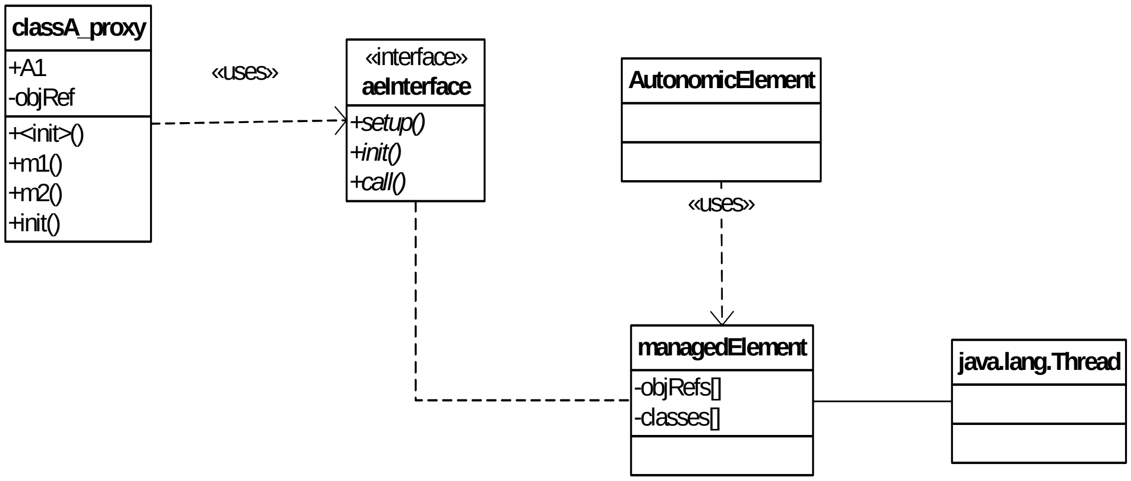


Figure 14. Transformed Runtime Program Structure.

the deployment of the object to an autonomic element. Once all necessary information is gathered and processed, the proxy initially invokes the *setup* method of the corresponding *managedElement* with necessary information (such as class name for object instance, location of class and generated name for identifying that instance) as arguments. Next, the *init* method is invoked to create the actual object instance in the autonomic element with any initializing arguments. Any further method call from the proxy is then delegated by using the *invoke* method inside the proxy class. The overall sequence of managed element deployment is shown in Figure 15.

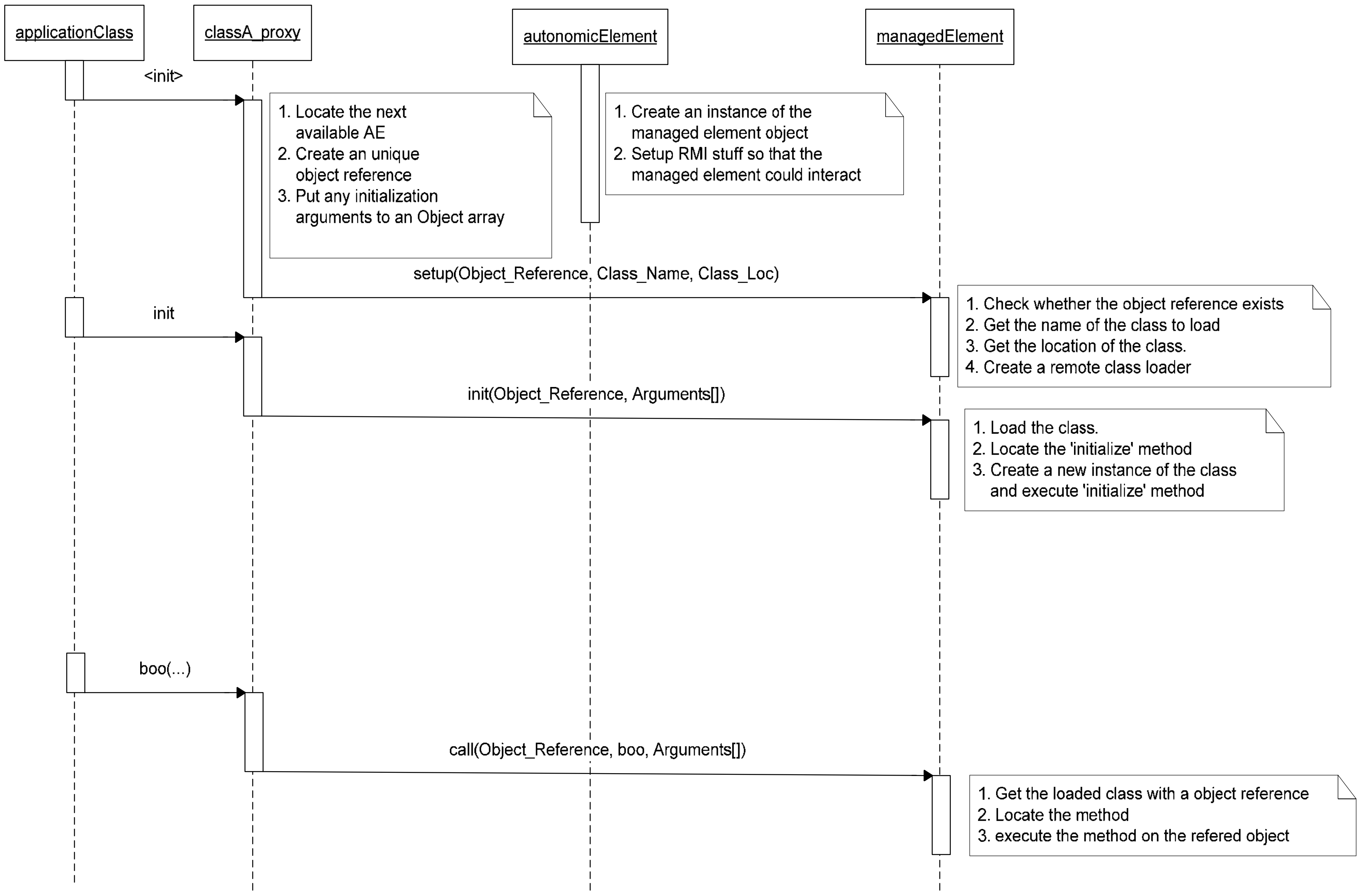

applicationClass
classA_proxy
autonomicElement
managedElement
<init>
1. Locate the next available AE
2. Create an unique object reference
3. Put any initialization arguments to an Object array
1. Create an instance of the managed element object
2. Setup RMI stuff so that the managed element could interact
setup(Object_Reference, Class_Name, Class_Loc)
1. Check whether the object reference exists
2. Get the name of the class to load
3. Get the location of the class.
4. Create a remote class loader
init
init(Object_Reference, Arguments[])
1. Load the class.
2. Locate the 'initialize' method
3. Create a new instance of the class and execute 'initialize' method
boo(...)
call(Object_Reference, boo, Arguments[])
1. Get the loaded class with a object reference
2. Locate the method
3. execute the method on the refered object

Figure 15. Sequence Diagram of Delegated Calls.

Figure 16 shows the timing for such remote calls, where other denotes the time it takes for local proxy class loading and locating the remote AE using the standard naming protocol. For a single remote call using the delegated proxy pattern, the increase in execution time is approximately 45%. With several calls to the same AE, this reduces to less then 10% as the JVM normally caches classes and for subsequent calls class loading and reflection occur faster than the initial time. Table 6 shows this behavior of the proxy based remote invocation.

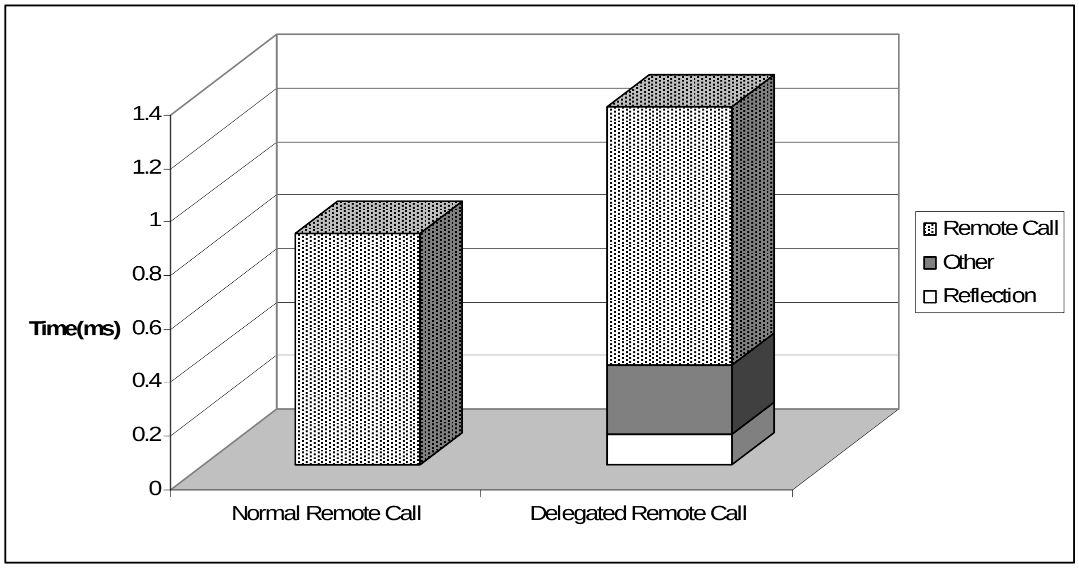


Figure 16. Timing Information for Delegated Invocation.

Table 6. Effects of Multiple Calls on Respnose Time.

Number of calls	Standard Remote Call (ms)	Delegated Remote Call (ms)	% Change
1000	391	476	21.74
10000	2289	2593	13.28
100000	19372	20261	4.59

The benefit of having a double delegated structure is that the distribution concerns and the self-management concerns are separated at the code level. This allows future code extension, since adding the two level delegations work in two separate processes and in sequence. Programmers can add their own transformations in the future if they wish following a similar approach; however the penalties for doing so are further increased code inflation and reduced response time due to extra code execution and class loading.

Summary

The software architecture of the system is presented and different aspects of it is discussed. Architectural choices have a profound effect on the capabilities of any autonomic system and affect many of the design decisions during its implementation. The presented architecture supports computational and data intensive centralized applications where the computation-to-communication ratio is significant. The approach to transform code to add distribution concerns and self-management concerns is also presented and evaluated in this chapter.

CHAPTER 6

AUTONOMIC ELEMENT ARCHITECTURE

Current research in autonomic computing suffers from the lack of a common definition of the basic autonomic entities. Defining and developing the basic autonomic entities and making it publicly available would greatly simplify prototyping of different self-management techniques and permit autonomic techniques to be compared and benchmarked. Although different aspects of autonomic computing are explored in isolation, the structural operation of an autonomic element itself has not been completely modeled. This chapter presents a self regulating design of an autonomic element in a distributed object environment. The goal of this architectural design is to provide an easy to program autonomic element which can be implemented in most domains with only minor modifications. Profiling and experimentation with this design shows that it is lightweight and performs smoothly without causing deadlock (since all internal communication and data structures are deadlock resistant) or producing excessive overhead.

Introduction

In order to cope with the increasing complexity of software management, the autonomic computing paradigm was introduced such that computing systems possess the capability of self-management. As computing systems continue to evolve to meet ever changing needs, dynamic computing systems which are self-manageable give business and scientific organizations the ability to automate their computing activities.

Unfortunately many programmers lack the skill to develop such self-managed systems due to the added programming complexity. It is tremendously beneficial to programmers if they do not have to be concerned with the programming complexity of implementing different issues related to self-management. A computing environment that hides the complexity of distributed programming and addresses such self-management issues automatically is certainly desirable. However, this sort of support should be provided without the need for programmer intervention or consideration other than the generation of policies to guide the execution process. The provision of all such self-management features is an unresolved problem [83] and there are no truly autonomic computing systems in existence at this time.

Although every aspect of autonomic computing is a significant research challenge, it is necessary to commence designing and developing tools and methods towards the goal of autonomic computing. While different aspects of autonomic computing have been explored in isolation, there is a lack of effective autonomic infrastructure that enables programmers to develop and integrate their programs using autonomic primitives. Although there is general agreement on the structure of the autonomic element, there are no general implementation details of the internal architecture of the autonomic component. The lack of a common implementation of this basic autonomic component hinders the interactions among different self-managed technologies. Since most other research employs proprietary techniques to develop such components, it is difficult for programmers to incorporate autonomic properties in the design and development of such systems. The programmer is left still more concerned with learning and implementing such proprietary techniques rather than concentrating on the actual problem in hand.

This chapter presents an architectural design of the standard autonomic element and simulation in a realistic environment. Having an open and flexible structure for the most basic component of any autonomic system alleviates programmers from the complexities associated with designing, developing, integrating and managing autonomic systems. Design goals of the presented structure of the autonomic element are as follows:

1. It should be transparent to use and programmers should "program as usual" with minimal constraints.
2. It should employ open standards and common metaphors to develop the autonomic element.
3. It should be lightweight and should not consume system resources unnecessarily.

This chapter presents a Java-based autonomic element design that greatly simplifies prototyping of self-management techniques. Although this design is based on Java, any programming language that supports standard object oriented metaphors can be easily used to develop the equivalent model of the autonomic element. The design allows different techniques to be compared and helps investigate the interaction among different techniques. The goal of this architecture is to incorporate it into the autonomic infrastructure to assist the rapid deployment of user applications and easy-to-use transparent, self-management of distributed systems.

Architecture

Autonomic elements are the heart of any autonomic system. An autonomic element is described by two distinct parts. The autonomic manager provides the functional abilities of the element and the managed element is the entity that the autonomic manager is monitoring and controlling. All autonomic elements have a control loop (MAPE-Monitor, Analyze, Plan, Execute) that dictates the work flow of the different sub-components of the autonomic element. Figure 17 shows the design for the autonomic element.

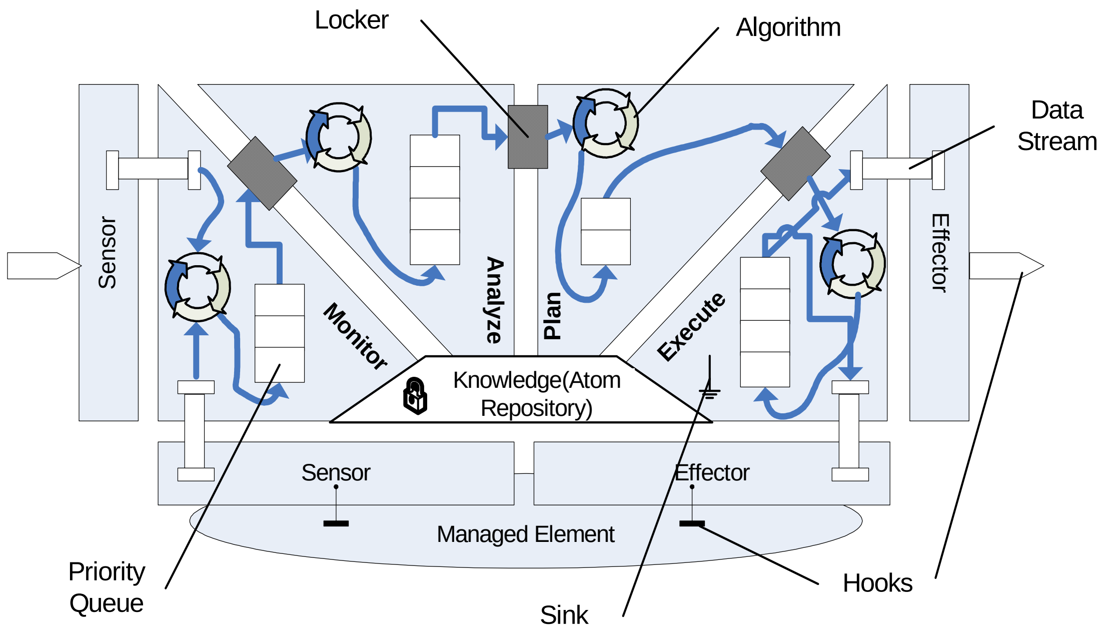


Figure 17. Internal Architecture of the Autonomic Element.

An autonomic element is envisioned as a multi-threaded server (daemon) having multiple components (monitor, analyze, plan and execute) and interfaces (sensors and effectors) running concurrently for better CPU utilization. Another alternative is to design the sub-components as threaded objects and schedule them according to a control loop.

As Java does not support language level thread control (stop, suspend etc. are deprecated as of Java 1.2), it is not possible to write a scheduler for Java threads without sacrificing performance. Writing such a scheduler is a complicated programming task and makes the autonomic element bulky and consumes more CPU power for scheduling. Instead, this thesis proposes each sub-component as a threaded object and delegates the scheduling decisions to the underlying JVM. The benefit of having a threaded model for the MAPE control loop over a sequential loop is:

1. Better CPU utilization by concurrently executing the sub-components. Since multi-core and multi-processor machines are more affordable and common then ever a concurrent MAPE loop allows autonomic elements to respond quickly to environment changes.
2. Algorithms which are responsible for different aspects of self-management can easily be incorporated as a child thread inside any of these parent threads denoting the four major components of the autonomic element.
3. Better throughput for information processing as no sub-component will be blocked waiting for another slower processing sub-component. Having a sequential control loop can block the execution of the whole element when one component is blocked for any reason and therefore may have an adverse performance effect.

To have the same semantic functionality as the control loop, this design uses semaphores in strategic places to order the execution of the sub-components and synchronize the threads when accessing shared data. This way, the control loop in this design is non-blocking due to concurrent execution of the individual sub-components.

This allows faster execution of autonomic behaviors but introduces the following challenges:

1. Asynchronous execution: Since each component is running asynchronously, it has to be guaranteed that no internal messages are passed inadvertently to a different component.
2. Deadlock and starvation: Care must be taken during development of shared data structures and data components to ensure that access to those shared resources are mutually exclusive. This is accomplished by using semaphore in strategic places to control concurrent access to shared data.

There are separate threads for environment interfaces (sensors and effectors) and for the main components (monitor, analyze, etc.) of the autonomic element. From time to time, the autonomic element has to perform its own management and accounting tasks. So the design is extended beyond the standard notion of autonomic elements with an element manager and several control interfaces. As shown in Figure 18, the autonomic manager assumes the role (*setup* or *active*, see Life Cycle Section) that is being required of it. Some of the autonomic elements in the system are allocated some higher level administrative authority. These managerial autonomic elements will either manage system registry and policy depository, or will act as the user interface for program partitioning and transformation, monitoring, or as the source or destination of program input and output. However, all the autonomic elements in the system have the same properties and they could act in any of the above roles if they are instructed to by other autonomic element.

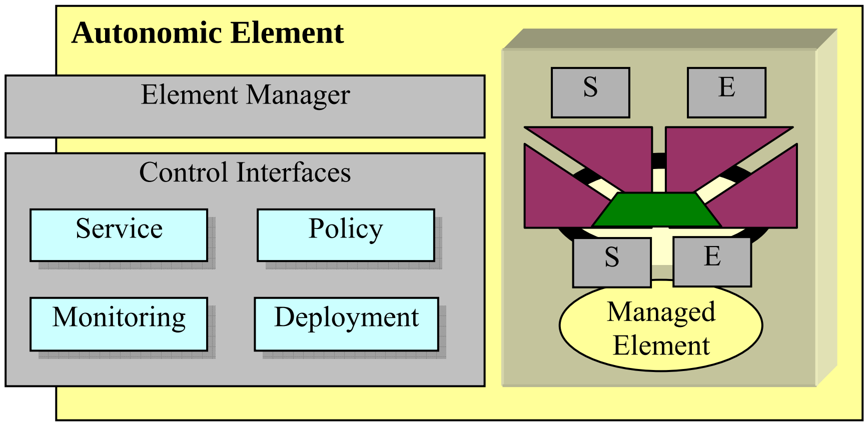


Figure 18. Autonomic Element Extensions.

There are multiple interfaces for the different services to be described, discovered and supplied. For instance, the *Service* interface allows other autonomic elements to reach a service agreement with an autonomic element. The *Policy* interface provides a way to transfer and modify policies between different autonomic elements. The *Monitoring* interface provides methods to monitor each autonomic element's internal activities and status information. The *Deployment* interface provides methods through which managed elements are allocated, deployed or restarted. Separating the functional aspects of autonomic management from the management of the autonomic element itself makes the overall software architecture more modular and extensible.

Life Cycle

The life cycle for the autonomic element is shown in Figure 19. Once the autonomic element is initiated, it moves into the setup state. In this state, there is no managed element attached to the autonomic element and the autonomic element is acting basically as a bootstrap manager for future deployment of the autonomic element with a

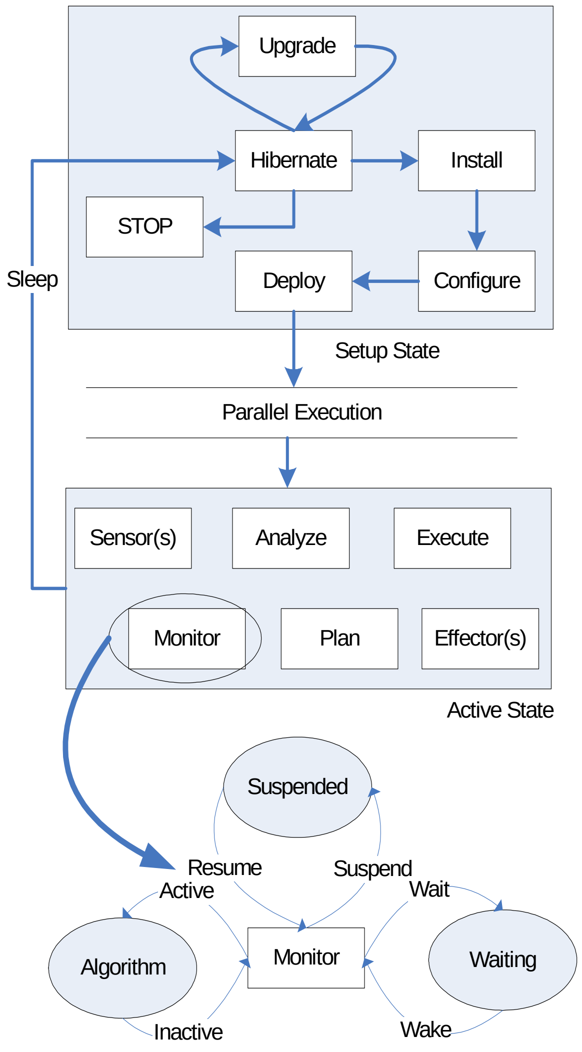


Figure 19. Autonomic Element Life Cycle.

managed element. In this state, the element remains in the hibernate state until either:

1. *The autonomic element is upgraded with a new version*: Each of the sub-components of the autonomic element is implemented using a technique described in [73]. Explicitly, the functional part of a sub-component is implemented through a common interface so that at runtime it can be updated, if necessary, to fulfill user policy and goals. The functional code resides inside a class (*task.java*) which implements a common interface named (*taskInterface.java*). At runtime, the old task class is

replaced by any newer version of its implementation. This allows adaptive composition of the component by adding, deleting or modifying the algorithm of each sub-component during run-time. This provides a general framework for the activities needed for the self-management of systems without forcing a specific programming model for the implementation of the actual operations. For instance, the programmer now has the freedom to implement the plan algorithm as either an artificial intelligence planner or a graph-based solution leveraging domain-specific knowledge and can switch between multiple algorithms during run-time without jeopardizing program execution.

2. *Autonomic management of a managed element is sought and the autonomic element switches to the active state*: If such a request is received, the element manager first creates all the necessary threads for the autonomic elements (*install*) to deploy. The threads are then configured with any initialization values (*configure*) and their execution is initiated (*deploy*) and the state of the autonomic element is moved to the active state. In the active state, all threads are subject to the underlying JVM thread model and can have multiple internal states as the platform dependent implementation of the threads. If the managed element is no longer to be managed by the autonomic element, it is switched back to the setup state and remains in the hibernate position until a new phase begins.
3. *The autonomic manager receives a stop signal from the administrator to stop execution.*

The setup state executes in a sequential fashion to guarantee that the sub-states follow a pre-determined path of execution. Whereas the active state follows a parallel execution, where each of the sub-states follows its own execution path determined by the underlying thread scheduler and communicates with each other via synchronized semaphores.

Implementation Details

Prior to describing the implementation of the components, the next section (Data Structures) describes the data structures used to create the internal formation of the autonomic element and the section after that (Operation of the Sub-components) describes the working of the different sub-components.

Data Structures

The choice of data structures has a profound impact on the performance of the element. Use of efficient data structures ensures smaller memory usage and faster execution of the element as a whole.

Data Atoms. All internal communication is performed in a standard format, named atoms and molecules. Atoms normally travel between different components in the direction of the control flow. An atom is essentially a collection of XML tokens which can be interpreted by all the sub-components of the autonomic element. The main reason for selecting XML to represent internal data is that it gives flexibility for future improvement and extension. Since XML is machine agnostic, autonomic elements developed in other programming languages can be incorporated easily into an existing system. Data atoms have the capability of chaining *n* atoms together to form a single molecule. Figure 20 shows the internal structure of a data molecule. Atoms need to chain to each other when the occurrence of certain events requires a steady supply of atoms which have some form of common relationship. However, care needs to be taken with respect to the size of a molecule (upper bound for *n*), as making an excessively sized data molecule can slow down the autonomic element. The self-optimization aspect of the autonomic element can dynamically change the size of a molecule (value for *n*) by monitoring the element over long runs.

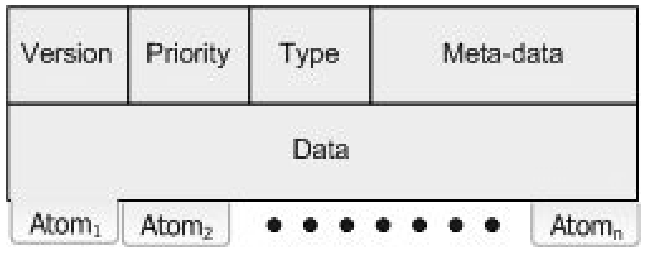


Figure 20. Structure of a Data Molecule.

Priority Queue. Each of the components inside the autonomic element has a buffer implemented as a priority queue. As data molecules are flowing through the control loop, they are stored temporarily inside the components for processing. Different components can create new data molecules after processing existing data atoms and can pass those to the next component. This design permits at most one molecule to pass between components at any given time. This ensures a consistent flow of information among the components and ensures that individual components are not overwhelmed. Therefore, molecules are sorted and stored in the queue based on their priority and are processed according to the highest priority. Normally the size of the queue starts with a predefined size. However, over time it may grow or shrink depending on the flow of molecules in the system.

Locker. This is a special data structure that resembles an operating system semaphore with added functionality (Figure 17). Lockers are placed between each pair of sub-components in the autonomic element to transfer atoms between them. The main responsibility of the locker is to receive an atom when it is empty and notify the destination component of an incoming atom. Although it is desirable to keep the communication mechanism open between the individual sub-components from an architectural standpoint, lockers are introduced to pass information (data molecules) between components for two main reasons. Firstly, to avoid any deadlock situation between two concurrent components during data transfer. Secondly, to avoid wasting any processing time of the components unnecessarily. Instead of the components polling each other for molecules, the locker notifies the corresponding component if there is a new molecule awaiting processing. Polling introduces synchronization and deadlock issues and a component is expected to spend most of its allotted schedule time polling for incoming atoms. Having all atoms transfer through the locker improves the response time of the components and does not block the receiver if the sender is processing at a slower rate than itself for any particular reason. All lockers fulfill the following two conditions:

1. The locker is always unidirectional. Therefore, only one component can push a molecule in a locker and only the adjacent component can pop it. To loop an atom back to a previous component, it has to passed around the control loop and appear in the queue of the destination component. It is possible to have a bi-directional locker between components; however, care should be taken to avoid any possibility of deadlock. A bi-directional locker does not increase performance, rather complicates the whole design. In the current version of the design bi-

directional lockers are not permitted, but with extended behavior modeling and verification, this could possibly be implemented; but no significant gain is expected from doing so.

2. As with semaphores, the push and pop operations must be atomic. Since Java does not provide any language level abstraction for atomic methods, we used the atomic reference object implementation in *java.util.concurrent.atomicReference* of the Java 1.5 API to implement those methods. So the methods have an atomic object reference, where the data atom can be pushed or popped atomically. The use of this class of variable offers higher performance than is available by using standard synchronization techniques.

Data Streams. These are one-way dedicated communication channels between two entities. Only one thread can write into it and another can read from it. To avoid any deadlock situation, all read and write operations are forced to be atomic as described earlier. Data streams are used where no intermediate processing is required, such as in the sensors and effectors to provide a rapid response.

Hooks. This is a collection of data structures and methods that allow the sensors and effectors to actually interact with the environment or the managed element. The representation for such hooks may not be universal because of the diversity of applications and environments. The environmental hooks can be as simple as network level sockets with assigned ports for receiving incoming atoms and sending outgoing atoms. Hooks for managed elements may require programming the managed elements through predefined interfaces such that at run time there are predefined methods that can be called by the sensors or effectors. However, this places extra responsibility and a burden on the programmer to preserve self-managing aspects. The development of such hooks is domain specific and should be addressed on a per system/application basis. For the application domain utilized in this thesis, byte code segments are injected at strategic places in the distributed Java objects which are treated as the managed element. This sort of code injection is completely transparent to the programmer and performed during preprocessing and runtime and before deployment with an autonomic element.

Atom Repository. Any processed molecules that the system wants to store for future use are stored in the atom repository. This is identical to the knowledge part of the autonomic element. Along with storing atoms, it can also store rules and policies regarding different functional aspects of the autonomic element. Finding a good knowledge representation is a separate research challenge. This design used ACPL to represent rules and policies as it provides a user friendly form of policy definition, policy management and appropriate tools through an API to work with policies. Since ACPL is based on XML, this permits atoms to be seamlessly incorporated into the repository. Although multiple components can read simultaneously from the repository, only one component can write to it at any time. This is to ensure consistency of the data in the repository.

Operation of the Sub-components

The functionality of different components of the autonomic element and their structure is considered in more detail in this section.

Sensors and Effectors. There are two sensors in every autonomic element. One is responsible for interfacing with the environment and other is necessary to interface with the managed element. The common responsibility is to acquire runtime information and incoming messages from the managed element or environment respectively. The sensors are not heavyweight processes as they only check the validity and format of any incoming message and forward them to the monitor component. On the other hand, effectors receive data from the execute component and verify its format and then either forward it to the environment or invoke appropriate methods inside the managed element to effect its execution.

Sub-components. Usually each component takes molecules from its input source (locker or sensors) and processes it and puts it in the priority queue. At each scheduling cycle, it checks whether there is anything in the queue which needs further processing or needs to be passed to the next stage of execution. If the corresponding locker or effector is free, it then pushes the topmost data atom to the corresponding destination. Data molecules may not propagate all the way to the last component (*execute)* if it is internal to the autonomic element or there is no further processing available or necessary for a particular data molecule. Therefore the *execute* sub-component has a data sink point, which is basically an object nullifier. The garbage collector is called to free up the memory of those nullified objects. Since the Java garbage collector blocks all threads during garbage collection, calling it frequently decreases the performance of the autonomic element. So different heuristics (such as waiting until the number of nullified objects reaches a minimum threshold or the queue size reaches a specific length) are used to decide when to call the garbage collector.

Performance Analysis

A test bed environment is developed to simulate the behavior of the proposed autonomic element in a real environment with different parameters, such as input rate and the size of atom. This allows observation of how components interact with each other to be made and how much time they really spend for their own management to be measured. In executing the autonomic element, the following techniques are used to optimize the operation:

1. Classes for the components are designed in a modular fashion and class loading is performed only after determining that a particular module is needed. This provides a fast startup and keeps a small memory footprint when the element is not used. The drawback is that, response time may increase as classes have to load dynamically at runtime. However with proper class caching techniques and designing effective class loaders, this effect can be minimized.
2. Management of atoms (sorting, duplicate matching, updating, deleting, etc.) in the repository is only performed when the element is idle.

All experiments were performed on an eight processor (900 MHz each) SPARC Sun server running Solaris 9. A sequential version of the autonomic element is also implemented to compare the concurrent version against. In both cases, the MAPE loop has the same amount of processing on the intermediate atoms. Running both versions of the autonomic element in a single processor machine provides the sequential version a significant performance boost as there is no overhead related to thread scheduling. Since machines with multiple processors or cores are becoming increasingly common now-a-days, the concurrent version can exploit this architecture. Therefore, from our viewpoint, it is sensible to measure throughput over a fixed amount of time in both versions instead of measuring execution time of an atom or a number of atoms. To measure throughput, both versions are ran for a preset amount of time and are both flooded with data molecules at the same input rate. It is measured how many atoms both versions of the control loop can handle within that set amount of time. Figure 21 shows the number of atoms that both of the versions can handle for different data molecule sizes. The *y*-axis in Figure 21 is represented using a logarithmic scale to better signify the difference for smaller values. It is evident that the proposed concurrent version has a higher throughput than the sequential version, especially when the size of the data molecule increases substantially. On average, the concurrent version processed 57% more information than the sequential version in a fixed amount of time.

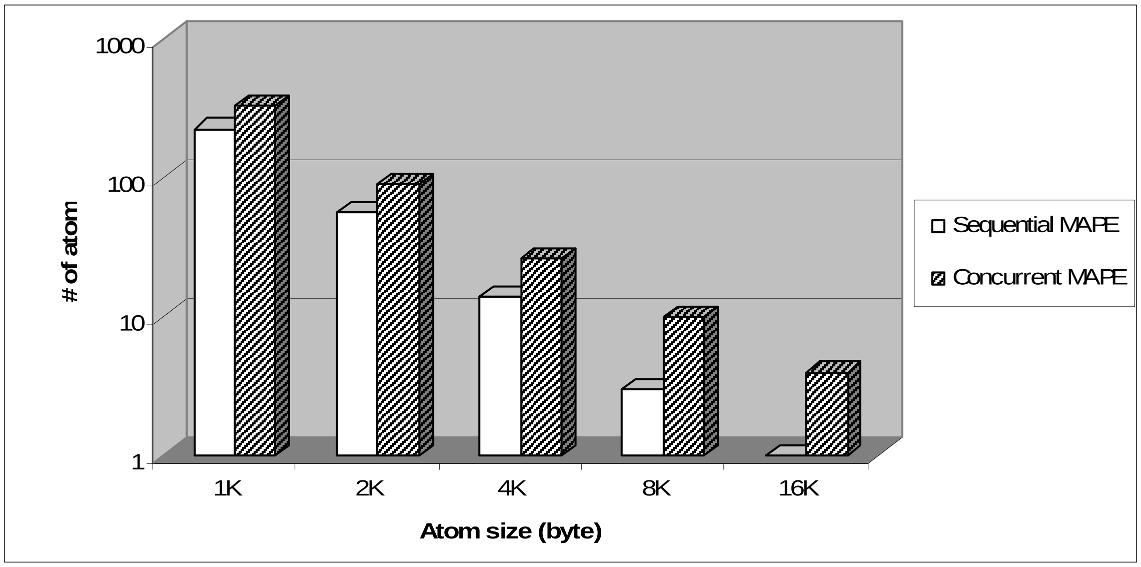


Figure 21. Runtime Throughput.

To observe how the concurrent architecture behaves in the case of different atom sizes and input rate, the concurrent version is executed with different atom sizes and different input rates for a fixed period of time. Figure 22 shows the throughput of the concurrent model in the case of various input rates and atom sizes. From the figure, it is obvious that the model is performing as expected by handling more atoms when the atom size is small and the number of atoms entering the system is high. The throughput decreases as the atom size increases and the input rate decreases. With 16K atom size, the

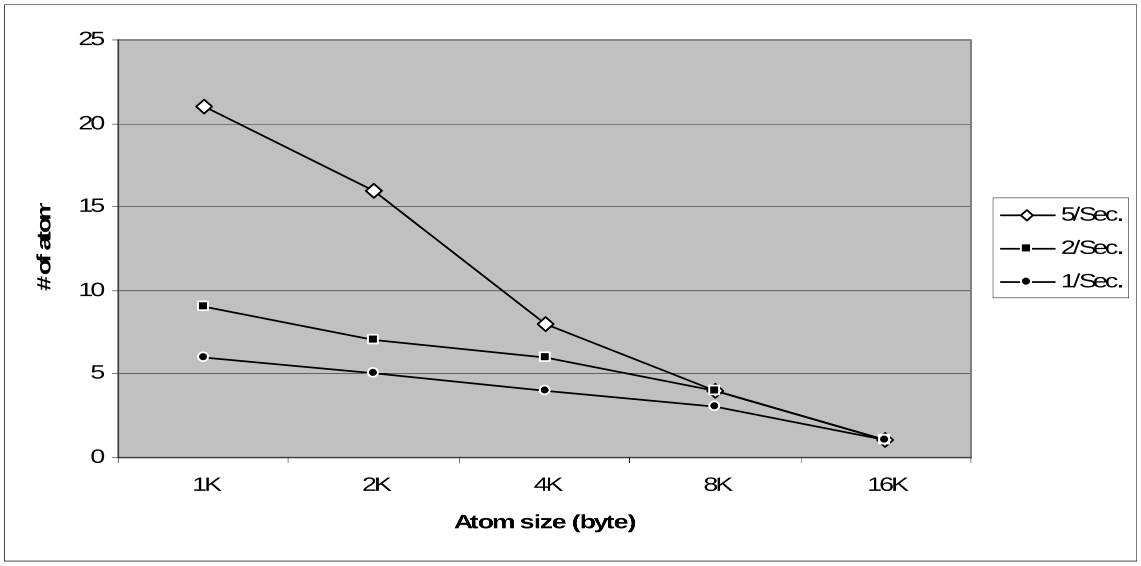


Figure 22. Flow of Atoms.

throughput becomes independent of the input rate as our fixed period is insufficient to handle such large atoms.

Table 7 shows different statistics related to the two versions of the control loop. In the concurrent version, the number of source code files (required to implement the architecture) may reduce if dynamic composition is not required, however in real-life situations it is inevitable, and we have to be comfortable with the added code inflation and class loading time. Both versions of the control loop are executed with the same input rate, atom size and for the same amount of time and are profiled using the NetBeans profiler [69]. Since the sequential version has a smaller number of runtime threads than the concurrent version, the maximum amount of heap used during the runtime is lower than the maximum heap used by the concurrent version. As a result, the sequential version spent 4.8% time less doing Garbage Collection (GC). Although this profile can change for different runs and for different parameters, the added cost for the concurrent version is worth taking for improved throughput, dynamic composition and better response time.

Table 7. Code Statistics.

MAPE Version	**# of source file**	**Total byte code**	**Profiling data**	
			Heap used	**GC**
Sequential	4	7806 byte	2.2 MB	16.3%
Concurrent	9	14820 byte	3.7 MB	21.1%

Summary

An autonomic element is the fundamental building block of any autonomic application and system. Although different aspects of autonomic computing are researched in isolation, the structural operation of an autonomic element has not been fully modeled. The standard definition for an autonomic element does not give a clear picture of how to build one from scratch and several proprietary designs have been proposed that are not interoperable with each other. This chapter presents an engineering perspective of building a domain independent autonomic element. It is important to have a well defined model of the basic building block to develop autonomic systems. The architectural design presented is self-regulating (in the sense that multiple internal sub-components are running in parallel but the internal data flow is consistent and unidirectional) and uses standard object oriented primitives and software engineering techniques to make it easy to develop and implement.

Analysis of runtime behavior and profiling shows that the design is sound and can work smoothly without causing any deadlock or producing extensive overhead. Future works include the implementation of the same design with other programming languages and incorporation in a system where autonomic elements from different programming language are in play. Once a stable and robust implementation is achieved, producing an API for an autonomic element would assist programmers to design autonomic elements in a consistent and standard form.

CHAPTER 7

ADDING SELF-HEALING

Adding different autonomic properties (self-configuring, self-healing, self-optimizing, self-protecting) into existing applications is immensely useful for redeploying them in an environment other than they were developed for. Such transformed applications can be redeployed in different dynamic environments without the user making changes to the application. However, creating such autonomic software entities is a significant challenge because of the amount of code transformation required and it will be tremendously beneficial to developers of such systems if such code transformations are performed automatically. This chapter presents techniques of injecting Java byte code with self-healing primitives to transform it to become a self-healing component.

Introduction

An autonomic self healing application should be able to recover from potential faults and should continue to work smoothly in the presence of such faults [58]. In the past, self-healing applications were rare and mostly confined to domains such as space craft control software, where taking a system down to correct faults was not an option. However, more and more of today's ever complex and large distributed software systems have the same requirement. For a large computational task, which is running for an extended period of time, it is not desirable or acceptable to abandon an execution due to a mundane fault. A system administrator could certainly fix most faults manually by analyzing logs and error reports, but that requires the system administrator to spend a large amount of time solving each individual fault. Furthermore, once the fault is identified and manually solved by the system administrator, the task has to be restarted from the beginning (or manually saved checkpoints), resulting in loss of useful computation and valuable time. However, with well defined policies and pre-defined goals, the software should heal itself in such situations and continue running transparently without any loss of valuable computation. This will not only save time and money in long term, but will also make the whole system more productive and responsive to environmental changes. The traditional approach for having such self-healing capabilities (or traditional fault tolerance) in an application is to build the application in such a way to either have those functionalities built into it (hardwired) or have an underlying framework to support such self-healing functionalities. Both of these approaches require that programmers of such systems know about specific details of their code and that the source code is readily available for modification after any runtime faults are identified. In reality this situation is often not the case as the original system developer may not always be readily available to interpret and solve every failure situation. For a new programmer or system administrator of a particular system, it becomes daunting to understand the functionality of a significant system and also interpret any faults produced by that particular system. In turn, it becomes nearly impossible for a new person to add transparent self-healing functionalities to such systems. It would be extremely beneficial if an existing application could be transformed and retrofitted with self-healing primitives so that the programmers do not have to deal with the self-healing and the fault management issues.

The technique presented in this chapter injects self-healing capabilities into byte code without any user involvement so that at run time, the application can heal itself after a limited set of software faults. Following are the design criteria under consideration in developing this approach:

- The target application should be written in Java; however, the technique could be implemented to other Common Language Runtime [66] based interpreted languages, such as C#. Since this technique works on a per method basis, programs written using object-oriented paradigms can be supported by this technique.
- No source code is required, all the self-healing capabilities are injected at the byte code level statically (using static byte code analysis) and dynamically (using runtime reflection). Since the system can work with byte code, programs whose source code is available can still be used just by compiling them to byte code with any standard Java compiler and then applying this technique to the byte code.
- The application programmer does not need to worry about low level checkpointing issues. Unlike other approaches [60], where users need to write programs in a specific way in order to accommodate checkpointing, this approach does not require any such construction. Users can manipulate the check pointing parameters with the use of high level policies.
- Any transient failures [13, 88] (exceptions) from the application are self-healed. If the failure can not be healed (such as a non-transient failure), the application is terminated in the normal fashion and the system administrator is informed. Logs and the checkpointed status information of the application are saved for the

administrator to diagnose such unsettled failures and once manually fixed, the program can be restarted using saved checkpointed status data.

Self-healing Architecture

The targeted application is instrumented with self-healing primitives and also with sensors and hooks at strategic points, so that at runtime, it is possible to interact with the managed code. Figure 23 shows the architecture of this approach. During runtime, the injected sensors are responsible for saving the current status information (SI) of the application at certain checkpointed locations. Depending on user given policies, sensor placement can be controlled. When any runtime failure occurs, the sensors also collect the context information related to that failure and passes that as failure information (FI) to the fault analyzer. The fault repository holds models and information for some of the

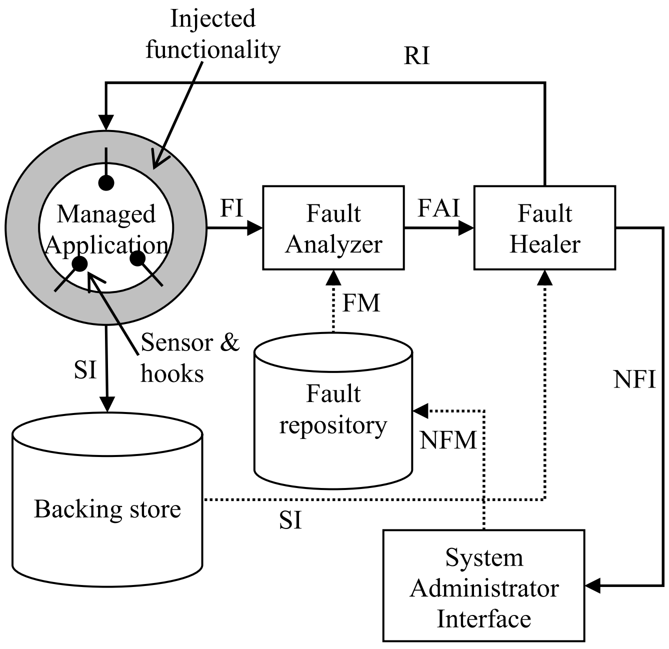


Figure 23. Self-healing Architecture.

most frequently occurring failures. The fault analyzer uses those fault models (FM) to analyze the runtime failures.

Once analyzed, the fault analyzer information (FAI) is passed to the fault healer for further processing. The fault healer takes the FAI and uses the last consistent checkpointed status information (SI) and tries to reconstruct the faulted method, so that it can be restarted at the point where the failure occurs. If the fault healer is successful in creating such recovery information (RI), it is passed to the managed application for recovery. Otherwise, the system administrator is notified of the new failure information (NFI) and the feedback is then saved in the fault repository as a new failure model (NFM). Once the new fault information is added to the repository, the fault healer will again try to heal the fault and will try to restart the application as previously described.

Faults and Fault Model

The approach presented in this chapter is concerned with faults that occur after the program is deployed. Such faults could result from problems or bugs in the user code, in the underlying physical node or network connection or in the run-time environment. Faults caused by bugs in the user code (logical errors), user generated custom exceptions or faults generated due to the functional aspect of the program are outside the control of this approach and should be addressed by the system administrator or the developer of the user program. A fault can be either transient (network outage, memory overload, disk space outage etc.) or non-transient (bug in the user code) [13, 88]. The technique presented in this chapter can self-heal such transient faults as once the system starts after self-healing, the condition that caused the fault will be healed and will not reappear.

However, non-transient faults are generally caused by some bug in the application code or due to unhandled exceptions. Although some types of non-transient faults can be self-healed, this may change the original semantic of the program or the fault will occur again as the condition that is creating the fault actually in the user code.

Each recoverable fault is represented internally by a four-tuple fault model FI = <F_{name}, F_{index}, F_{stack}, O_{info}>, where,

F_{name} = Name and type of the fault.

F_{index} = Byte code index where the fault occurs. This must be deduced from the byte code line number attribute [50] inside the class file as the JVM only gives the source code line when throwing an exception.

F_{stack} = Stack trace of the exception, transformed into chained fault models.

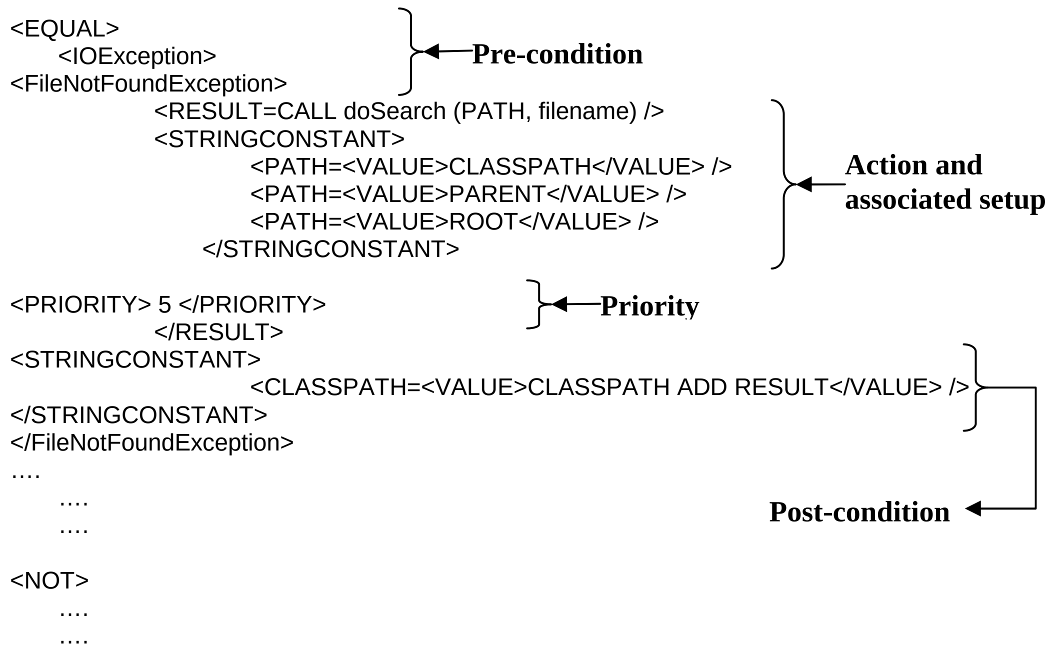
O_{info} = Object information such as, object and method name and formal and actual parameters of the method that caused the exception.

This representation allows the system to manage faults more easily and effectively. Representing faults in an accurate and consistent manner is not enough; the associated actions also have to be represented in some open standards so that the model can be extended or updated without the need to modify the underlying algorithm. Table 8 shows a subset of faults that this technique can heal. As shown in Table 8, there can be multiple possible actions for any specific fault. The number of such possibilities grows or reduces over time as the system adapts with feedback from the system administrator or through learning by reinforcement ($Action_{xn}$ did not heal $Fault_x$ for n number of times, so reduce its priority or disable the action altogether). It is possible to deploy different statistical and machine learning algorithms to learn new fault scenarios; however, it is out of the

Table 8. Faults and Default Actions.

Faults (Exception/Error)	**Possible Solution (s)**
ClassNotFoundException	Heuristic 1:Extend the class path. Heuristic 2:Search the file, locate it and then re-load it. Heuristic 3:Use custom class loader.
FileLockInterruptionException	Heuristic 1:Make the current thread to wait random amount (with an upper bound) of time to acquire the file lock.
UnknownHostException or *CommunicationException*	Heuristic 1:If the network is down then wait a random amount (with an upper bound) of time and try again (*n* number of times). Heuristic 2:If network is not down, then use ICMP messages and try to locate the host manually. Heuristic 3:Retransmit and re-route packets by delegating all calls to a custom network level delegation manager.
NoSuchMethodException or *NoSuchFieldException*	Heuristic 1:Check whether the correct version of the class being used to do runtime reflection.
ServerNotActiveException	Heuristic 1:If the network is down then wait a random amount (with an upper bound) of time and try again (*n* number of times). Heuristic 2:If the network is not down then try to locate a replacement server for the remote call. Heuristic 3:Retransmit and re-route packets by delegating all calls to a custom network level delegation manager.
IOException	Heuristic 1:This is the super class of an array of IO related faults. Carefully analyzing the message thrown by this exception and the actual subclass that throws this exception, it is possible to devise several heuristics to heal such faults.
AccessControlException	Heuristic 1:Try to change the security level of the current thread by replacing the current security manager with a custom security manager.
BindException	Heuristic 1:Wait a random amount (with an upper bound) of time to see whether the intended address/port frees up. Heuristic 2:Try to use a new address/port and delegate all calls to the new address.

scope of this thesis. We adopt the Autonomic Computing Policy Language (ACPL) in describing policies of different fault scenarios. This not only allows us to address a consistent policy management across the system, but also provides a user friendly form of policy definition, policy management and tools to work with policies. The fault repository represents all fault information and actions using ACPL. Figure 24 gives a snapshot of the fault repository database that stores all fault models. All faults are described by a pre-condition (what to match), one or more actions with associated setup, priority of a certain pre-condition and any post-conditions. Priority is used to select among multiple actions for the same pre-condition. Assignment of priority to an action is a challenging issue and a simple utility based priority management algorithm is being used to assign priorities among multiple actions. Since ACPL is XML-based, future extension of this technique or integration of this technique into an autonomic environment is simpler for the application programmer.



```
<EQUAL>
     <IOException>
<FileNotFoundException>
               <RESULT=CALL doSearch (PATH, filename) />
               <STRINGCONSTANT>
                         <PATH=<VALUE>CLASSPATH</VALUE> />
                         <PATH=<VALUE>PARENT</VALUE> />
                         <PATH=<VALUE>ROOT</VALUE> />
                    </STRINGCONSTANT>

<PRIORITY> 5 </PRIORITY>
               </RESULT>
<STRINGCONSTANT>
                         <CLASSPATH=<VALUE>CLASSPATH ADD RESULT</VALUE> />
</STRINGCONSTANT>
</FileNotFoundException>
....
     ....
     ....

<NOT>
     ....
     ....
```

Figure 24. Representation of Fault Model.

Code Transformations

To add the extra functionality to an existing class required for the self-healing purposes, code transformation at byte code level is necessary. The byte code rewriting tool Javassist [15] is used to introspect and retrofit existing byte codes with self-healing primitives. To handle any runtime exceptions, an extra try-catch block is inserted in every method. At runtime, if any runtime exception occurs, it is caught and analyzed to find the cause of the exception.

All analyzed information is saved as logs, so that the system administrator can later inspect them and take appropriate action if necessary. Once such actions are taken, it is recorded to the fault repository so that next time the same fault can be avoided. A clone of the running method is created where, at the beginning of the code, new code is inserted to reinitialize the object with a consistent state and to restart the method at the point where the fault occurs. Since the rest of the code already has the injected self-healing mechanism, any further faults are handled in the same way as described above. The algorithm for this process is presented in Algorithm 1. The algorithm works on a per class basis and initially, it checks whether there is local variable attribute attached with the methods of the class. If the local variables attribute was originally not included with the methods, it is added during static analysis [24] using Algorithm 2. This algorithm verifies the existence of the local variable attribute before transforming the code. Since a new field is added through which the status information is checkpointed, it needs to be initialized in the constructors. Byte code sequences are generated and added at the end of any existing code in the constructor to avoid inheritance related constraints placed on the

```
1. If there is no local variable attribute with the methods then use Algorithm 2 to
   generate and attach the local variable attribute with the methods.
2. Add a new field in the class of type statusObject
3. For each constructor in the class, do
      a. Generate a new byte code sequence to initialize the newly added field in the
         class.
      b. Insert the byte code sequence at the end of any existing code in the
         constructor.
      c. Calculate the new stack depth for that constructor and change the maxStack
         attribute of the constructor.
      d. Encapsulate the body of the constructor with a new try-catch block that
         catches Throwable and has the fault analyser and fault healer code as the
         body of the catch block.
4. For each method in the class, do
      a. Generate a new byte code sequence to add a new Object array in the method
         as a local variable.
      b. Insert the byte code sequence at the top of any existing code in the method.
      c. Since a new local variable is added, change the maxLocals attribute and add
         the new local variable information in the local variable attribute.
      d. For all  byte code instructions in the method, do:
            i. If the current byte code is within a try block and the current byte code
               index is a checkpointable position, then
                  1. Get all local variables that have scope in that position.
                  2. Generate byte codes to wrap those local variables with
                     proper object types (if primitive) and then assign each of
                     them to different cells of the newly introduced Object array.
                  3. Generate byte code to call the checkpoint method of the
                     statusObject field with the Object array as one of its
                     parameter.
                  4. Concatenate these two generated byte codes and insert
                     them in the original code sequence at the point where it was
                     determined as a checkpointable position.
      e. Calculate the new stack depth for that method and change the maxStack
         attribute of the method.
      f. Encapsulate the body of the method with a new try-catch block that catches
         Throwable and has the fault analyser and fault healer code as the body of the
         catch block.
```

Algorithm 1. Self-healing transformations.

constructor (such as calls to *super* must occur first before any other statement). For methods, a new *Object* array is introduced as a local variable and the length of the array is set to the total number of local variables inside that method. Care needs to be taken in calculating the number of local variables inside a method since any arguments to that method are also counted as local variables (see Local Variables Section) and the nature of the method (static or instance) also dictates how those local variables are numbered. Therefore, the *maxLocal* attribute in the method is updated with the newly calculated number of local variables. At a checkpoint location, the byte code sequence is inserted that assigns each local variable with in the scope of that position to the cells of the *Object* array declared previously. However, the challenge is to wrap each of these variables (primitive types) with proper *Object* classes and assign them to individual cells of the *Object* array. The local variable attribute is used to determine each of the variable's type and then appropriate wrapper classes are used. For reference types (including arrays), nothing extra needs to be done since standard polymorphism takes precedence in such assignment. The call to the *checkpoint* method inside the *statusObject* class is then generated in the byte code and added to the sequence generated thus far. To pass all arguments to that method, each of the arguments are loaded first in the runtime stack in the same order as the formal parameters, before actually invoking the method. This is to ensure the correct semantics of the stack-based JVM execution model. The *Object* array, current method name, current byte code offset and the current object reference is passed as arguments to the *checkpoint* method. Adding new codes in existing methods (including constructors) introduces some new challenges and forces recalculation of the maximum stack depth during execution and consequently the corresponding byte code attribute (*maxStack*) needs updating. Without recalculating and updating this attribute, the resulting code will not be verified by the Java Runtime Verifier [50] and will not be executed by the JVM.

Figure 25 shows the transformations that take place during the lifetime of a method. Particularly:

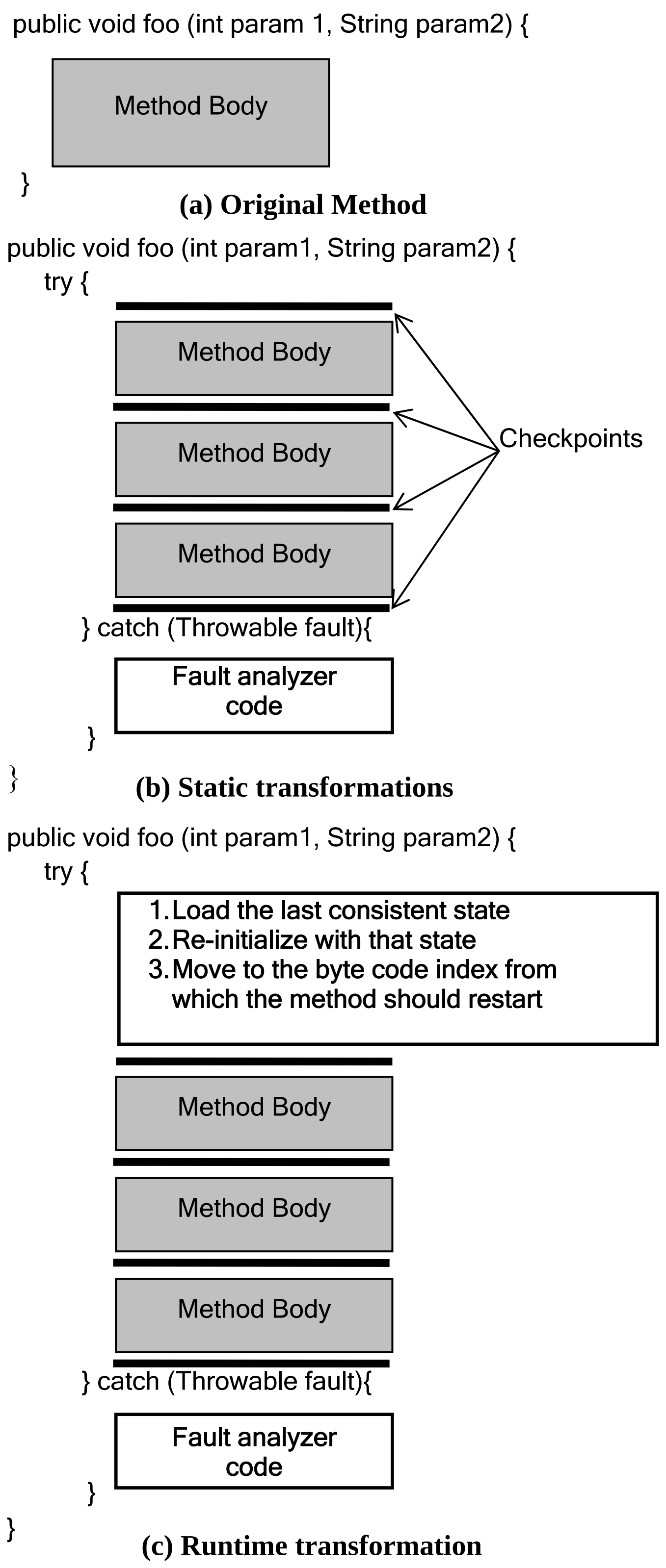

public void foo (int param 1, String param2) {
Method Body
}
(a) Original Method
public void foo (int param1, String param2) {
try {
Method Body
Method Body
Method Body
Checkpoints
} catch (Throwable fault){
Fault analyzer code
}
}
(b) Static transformations
public void foo (int param1, String param2) {
try {
1. Load the last consistent state
2. Re-initialize with that state
3. Move to the byte code index from which the method should restart
Method Body
Method Body
Method Body
} catch (Throwable fault){
Fault analyzer code
}
}
(c) Runtime transformation

Figure 25. Self-healing Transformation Example.

1. The original method body is wrapped in an extra *try-catch* block and checkpoint calls are inserted in the body itself.
2. If any exception is thrown by the code, it is caught by the newly inserted *try-catch* block, where the fault analyzer is inserted.
3. The caught exception is analyzed and the process starts to heal that particular fault.

Several adaptations are needed to support existing programs with the desired autonomic behavior, which are described in the next few sections in more detail.

Local Variables

To save the state information of any running method inside the object, local variable values have to be saved during checkpointing along with other necessary information, such as the line number of the next instruction to be executed, value of actual parameters and value of any object field. To gather the current state of the object, all the local variables in the current scope along with any fields and class variable's values need to be saved. As the Java compiler does not save any local variable information (such as type information or variable scope), the static analyzer gathers this information by analyzing the byte code and adds that information as a byte code attribute [50] to each method. During self-healing transformations, this attribute is used to create appropriate wrappers to save the corresponding object status. The challenge is to recreate a local variable table from the byte code when there is no information saved for the local variables. Byte code instructions access local variables from a zero based array in the runtime stack and depending on the type of method (class or instance), variables and arguments are indexed differently.

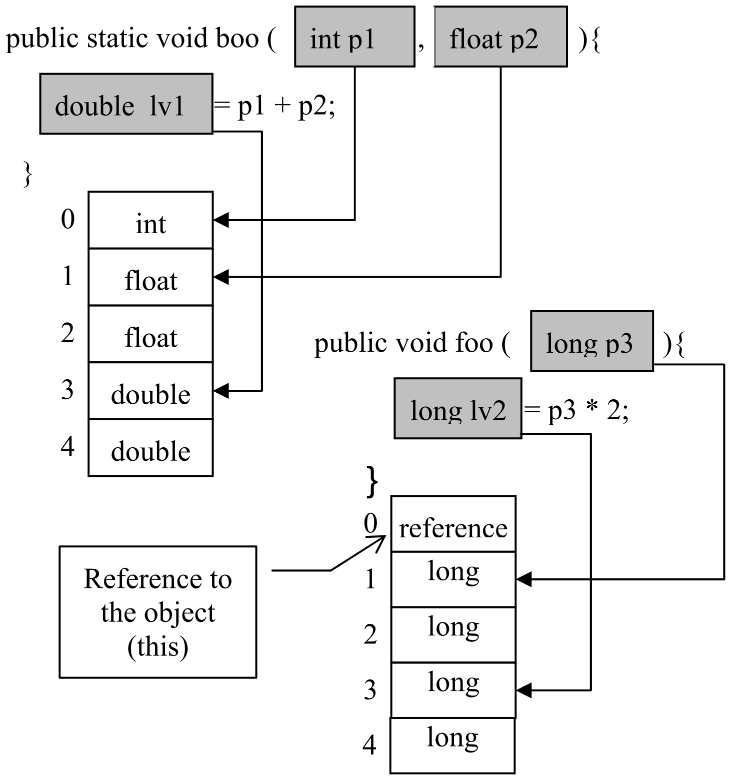


Figure 26. Local Variable Access Indexing.

As shown in Figure 26, along with all local variables, method parameters and object instance (only with instance methods) are also placed on that array. Depending on the data type, variables will occupy 1 or 2 cells in the array. Depending on the scope of the variable, cells can be reused by different variables within different scopes of the method body. The challenge is to represent such dual usage in a consistent manner so that the local variable can be accessed with a proper type cast at the proper location in the byte code. The algorithm adds a duplicate entry with the same index into the local variable table for such occurrences. Different scope information is used to distinguish between these two entries. Figure 27 shows the code segment which is responsible for adding local variable information into the local variable attribute. If there is no local variable information in the attribute, then the first variable is added directly. Otherwise, each variable in the attribute is compared with the given variable information and corresponding decision (same variable with an extending scope, different variable but using same index etc.) is made on how to represent that variable. 4 specific data element are saved for each variable: the start and the end byte code index of the variable's scope,

```
for(int i=0;i<numVar;i++){
          int varIndex=ByteArray.readU16bit(info,i*8+8);
if(varIndex==index){   // Whether the variable is already been added?
   alreadyAdded=true;
   int typeAtIndex=descriptorIndex(i);
   if(type>0){  // Reference type
      ByteArray.write16bit(length,info,(i*8+4));  // Update the scope then
      break;
   }else if(typeAtIndex==type){  // Identical (type) variable at identical index
      ByteArray.write16bit(length,info,(i*8+4));  // Update the scope then
      break;
   }else{  // Different variable at identical index
      //1. Change the scope of the other entry
      ByteArray.write16bit(length-1,info,(i*8+4));
      //2. Add the new entry
      ByteArray.write16bit(tableLength() + 1, newInfo, 0);
      // 2 bytes for each data
      ByteArray.write16bit(startPc, newInfo, size);      // Start of variable scope
      ByteArray.write16bit(length, newInfo, size + 2); // End of variable scope
      ByteArray.write16bit(type, newInfo, size + 4);     // Type of variable
      ByteArray.write16bit(index, newInfo, size + 6);  // Variable index
   }
}

if(!alreadyAdded){  // If the local variable haven't been added, then add it
    ByteArray.write16bit(tableLength() + 1, newInfo, 0);

   // 2 bytes for each data
    ByteArray.write16bit(startPc, newInfo, size);      // Start of variable scope
    ByteArray.write16bit(length, newInfo, size + 2); // End of variable scope
    ByteArray.write16bit(type, newInfo, size + 4);     // Type of variable
    ByteArray.write16bit(index, newInfo, size + 6);  // Variable index
}
```

Figure 27. Code Fragment to generate a New Local Variable Information.

the variable type represented by pre-defined numbers for each possible Java data type, and the index into the local variable array in the method (Figure 26).

To determine variable scope, variable index within the array and data types, Algorithm 2 is utilized (See Appendix A for the detailed flow chart). The time taken by Algorithm 2 is proportional to the number of local variables in the method. Only 8 bytes are used to save all necessary information about a local variable. This keeps code inflation to a minimum. See Performance Analysis Section for some comparisons. The algorithm requires only one pass over the code to determine the local variables and it is performed simultaneously during the static analysis [24], therefore there is no extra overhead for executing this algorithm.

```
1.  For all byte code instructions in a method, do
        a.  Get next opcode (and operand, if any)
            i.   If the opcode is of a data store type, then determine the
                 type from the opcode (for primitive type). For reference
                 (object) type, check the constant pool and determine the
                 type information.
            ii.  Determine the variable index from the opcode suffix or
                 from the operand. Also, determine the corresponding byte
                 code offset, where it is declared for the first time.
            iii. If the opcode represents an already identified variable with
                 the same data type, then update that variable's scope
                 information.
            iv.  If the opcode represents an already identified variable with
                 different data type (same variable index is used by two
                 variables in two different scope), then create a new
                 variable entry.
```

Algorithm 2. Generate Local Variable Information.

Since most opcodes in the JVM instruction set explicitly encode type information (for instance, opcode for integer type starts with ‘i’, opcode for float type starts with ‘f’ and so on) about the operations they perform, it is easy to determine data types of most variables by looking at the opcode itself. However, JVM uses integer data types to represent *boolean*, *char*, *short* and *byte* data types also and includes opcodes that convert from one primitive type to other. Therefore, care needs to be taken to determine the actual data type of a variable. Correctness can be guaranteed by looking for a type conversion opcode, before and after an opcode that is using a variable for the first time.

Adding Checkpoints

To restart seamlessly from a crash or after a runtime exception, the internal state (fields, method parameters and local variables, next byte code to be executed) of the object must be made persistent. According to a given user policy, instrumentation points in the code are determined and appropriate method calls are inserted to checkpoint the current state of the object. The following checkpoint locations are identified at the byte code level:

1. After several write operations.
2. Before and after an I/O interaction.
3. Before and after any data interaction between two objects.

The checkpointing algorithm below (Algorithm 3) describes adding a status object as a field into the target class, so that the checkpointed data can be saved through that field. Algorithm 3 provides checkpointing in byte code on a per method basis and uses different byte code attributes, such as *code*, *exception*, *line number* and *local variable* (generated by Algorithm 2) attributes. See Appendix A for the detailed flow chart of this algorithm.

Proper type casting is necessary to store and to restore checkpointed data. All the local variable and actual parameter information is type sensitive and requires careful type manipulation. All primitive method data is converted to its corresponding object format and saved as objects in the persistent data store. During restoration, data is cast back to its original type by gathering the type information of each local variable and actual parameter from the local variable attribute.

1. *Insert an array of objects at the top of the method body. The size of this array is equal to the number of local variables in the method.*
2. *Determine the next checkpointable position in the byte code.*
3. *Assign object array with local variables having corresponding index value.*
4. *Load this object array and all other status information in to the run time stack.*
5. *Call the checkpoint method, which takes all the status information from the runtime stack and saves that in a persistent store.*

Algorithm 3. Checkpoint a Class at the Byte Code Level.

Status Data

During runtime, whenever any checkpoint is encountered, the current status of the object is incrementally saved to a platform independent disk file or backing store. The implementation of the checkpointed data manipulation algorithm is performed by synthesizing and re-engineering ideas found in existing fault tolerant techniques for distributed applications. Using pre-defined policies, users can modify the behavior of such algorithms to suit their needs. For example, to save disk space, once an object completes its execution and is garbage collected, all the corresponding status information related to that object is deleted from the backing store. After a fault, determining a consistent state from which the program should be restarted is complex and requires extensive analysis of the runtime structure. However, to keep it simple and to demonstrate the viability of the approach, this thesis only considers resuming execution from the last checkpointed consistent state.

The implementation of the backing store where checkpointed data is saved is left open to the application programmer. The only constraint is that access to such backing store should occur through a standard interface provided by the system so that the self-healing algorithm requires no change in implementation. Currently, a backing store is implemented hierarchically (each domain has its own backing store) following the underlying networking structure and each domain (subnet) has its own backing store to cater for all object classes in that domain. To keep it simple and to provide faster response, the following adaptation is made in the current implementation of the backing store:

1. Objects of class *statusObject* (which holds the status of a method in a single checkpoint) are saved in a hash table keyed with the byte code index of the position where the checkpoint is performed. This is to easily identify checkpoints from the same method.
2. If the size of the hash table exceeds a pre-defined threshold, only then it is written to the underlying file. This is to minimize the time required for disk write operations. One possible downside is that, if the program faults before the hash table ever reaches the threshold, there will be no state saved in the backing store to restore from. In such a case, trying to read the last checkpointed state from the cached copy of the hash table may be the only option.

Task of Fault Analyzer and Fault Healer

The crucial part of the self healing process is performed by the fault analyzer-fault healer pair. The fault analyzer code is inserted at the byte code level, whereas the fault healer is a separate class structure that is initiated by the fault analyzer. The fault analyzer code inserted inside the method body is responsible for gathering the necessary code to create the failure information and then call the appropriate algorithm to analyze the failure information and compare that with the saved fault models to devise a plan of action for the fault healer. Once the fault healer obtains the required healing information, it performs the following steps to resume the operation:

1. Find the last consistent state (a consistent state is where all the threads were able to write their status successfully) and read the saved status data from the backing store. The backing store is implemented hierarchically across the network and after a fault, if the backing store in the application's current domain is not available, then the backing store in its parent's domain is accessed and this propagates in the worst case to the top of the hierarchical domain based backing store.
2. Create a clone of the method and discard all byte code sequences before the byte code index where execution must be resumed. The difficulty of creating such a cloned method is that not only the *maxLocals* and *maxStack* attribute of the resulting method must be calculated and updated, but also the local variable indices inside the existing byte code need to be updated. So, for instance if there is a local variable in the original method which is using variable index 0, for adding all the re-initialization code (Step 3), the index of that particular variables now should be updated (if *n* new

variables are added before that particular variable, its new index should be 0+*n*) to reflect correct class file semantics. So, wherever that particular variable was used by any opcode, it needs to be updated with the newly calculated correct index number. If this is not done properly, the method will crash during execution due to incorrect type conversion.

3. Inject byte code at the top of the method that reinitializes the internal data structures of the object with the status data. The last instruction in this newly added sequence is a *goto* instruction that directs the execution to the position in the method where we want execution to resume.
4. Start executing this cloned method using reflection and on the same object instance as the original method was running. Running the method on a newly created instance will create inconsistency in the program flow. Therefore to keep the same functional flow, the object instance on which the original method was executing is used to run the cloned method.

Normally, the fault healer saves all intermediate and cloned methods and associated byte code on the hard drive for debugging purposes. However, if the system administrator thinks that unnecessary, then it can be switched off and that will not only save disk space but also will speed up the healing process.

Runtime Exceptions

To handle any runtime exceptions, the method body is encapsulated within another try-catch block (which catches the Throwable super class) to give the method another opportunity to continue after the statement where the exception is thrown.

However, adding a new try-catch block introduces three different scenarios describe bellow, which need to be addressed at the byte code level to fully support runtime exceptions:

1. *Subclass of Throwable or Exception is already being caught by the method body.*

   Solution: Re-throw the exception so that the enclosing try-catch block can catch it and continue with its self-healing mechanism.

2. *There is already an encapsulating try-catch block that catches either java.lang.Exception or Throwable.*

   Solution: Instead of adding a new try-catch to enclose the entire method, the catch block is modified and the self-healing mechanism is added.

3. *A nested try-catch block already exists within the method body.*

   Solution: Re-throw every nested exception that is caught, so that the outer most enclosing try-catch block can catch it and continue with its self healing mechanism.

Since the rest of the code already has the injected exception handling mechanism, any recurring faults will be handled in the same way as described above. If the same fault is regularly occurring, the system administrator is notified to take proper action and the system will cease execution of that particular program.

While injecting byte codes for self-healing purpose or determining local variable information, any existing byte code sequence which is inside a user declared *catch* block is ignored. Since the goal is to keep the original functional flow of the program, there is no point of checkpointing inside an existing *catch* block or saving local variables declared in such blocks. Rather, the above mentioned policies to change or handle existing user declared *try-catch* block are used.

Performance Analysis

To judge the applicability of this technique in a real environment, traditional benchmarking techniques are insufficient. Simply measuring how fast a fault can be healed is dependent on the underlying platform and the runtime environment. Healing the same fault on two different platforms will result in different times due to different environment parameters. Providing only such a functional evaluation of the technique is insufficient to judge its effectiveness in the field of autonomic computing. Since one of the main goals of this research is to provide a transparent interface to the user, an evaluation framework to address other issues that can not be functionally evaluated is developed. Table 9 shows such an evaluation of this technique. As noted in [10], benchmarking an autonomic system is difficult and there is no widely accepted evaluation methodology. Since there are no other similar works to compare this technique with, this thesis measures the time taken to do such transformations, the resulting code inflation and the runtime memory usage inflation as metrics giving an insight into autonomic computing performance.

Experiments on the presented self-healing techniques were conducted with several Java programs (sequential and parallel) running on a single machine, having varying code structure and complexities (Appendix B). Program 1 writes random bytes into disk files to simulate IO failures. Program 2 attempts to write different data types on an open socket to simulate network failure scenarios. Finally program 3 implements a concurrent matrix multiplication algorithm to simulate synchronization and locking faults. The techniques described in this chapter can support Java byte code from JDK version 1.2 to 1.5. A standard PC equipped with a Pentium 4, 3.4 GHz CPU and 512 MB

Table 9. Evaluation Framework for the Technique.

Properties	**Abilities of this technique**
Usability	This technique is easy to use and programmers do not have to worry about pre and post processing.
Modularity	This technique is designed in a modular fashion so that it can be added or deleted from an existing system without any loss of original program semantic.
Transparency	It is completely transparent (all code transformation is done automatically) to the user and using this technique is straightforward.
Scalability	This technique adds self healing primitive on a per method basis. So other autonomic properties can be added in the same way without the need to modify this technique. However, with any such code injection, programmers have to be mindful of the resulting code inflation.
Persistence	This technique does not modify any data internal to the program itself. Hence the program execution remains unchanged functionally and semantically. Checkpointed status data is kept in a machine independent format using Java Serialization and saved in different portions of the data store to keep it consistent between runs.
Maintainability	Over a long period of time, the system can generate valuable usage data and execution pattern that will help software operators and programmers to maintain the code more effectively.

RAM running Windows XP is used to conduct all of the experiments described in this section. To generate the local variable information, algorithm 1 produces smaller files than using the Java compiler's debugging switch (-g during compilation). Since programmers may not use that switch, the self-healing code injector must rely on Algorithm 2, which, as shown in Table 10, performs well.

Table 10. Code Inflation Due to Local Variable Addition (in bytes).

Program	Original Size	Algorithm 1	javac –g
1	910	1036	1228
2	1208	1377	1639
3	5823	6155	7320

Experiments with two different forms of checkpointing policies were conducted. In the conservative policy, checkpoints are inserted after every statement in the method, whereas in the optimized policy, checkpointing decisions were made as described in Section Adding Checkpoints. The resultant code has a *O(n)* code inflation, where *n* is the size of the original code. Figure 28 shows the overall code inflation due to the added functionality of the code. On average the resulting code has around 51% code inflation. With better optimization techniques, this can almost certainly be lowered.

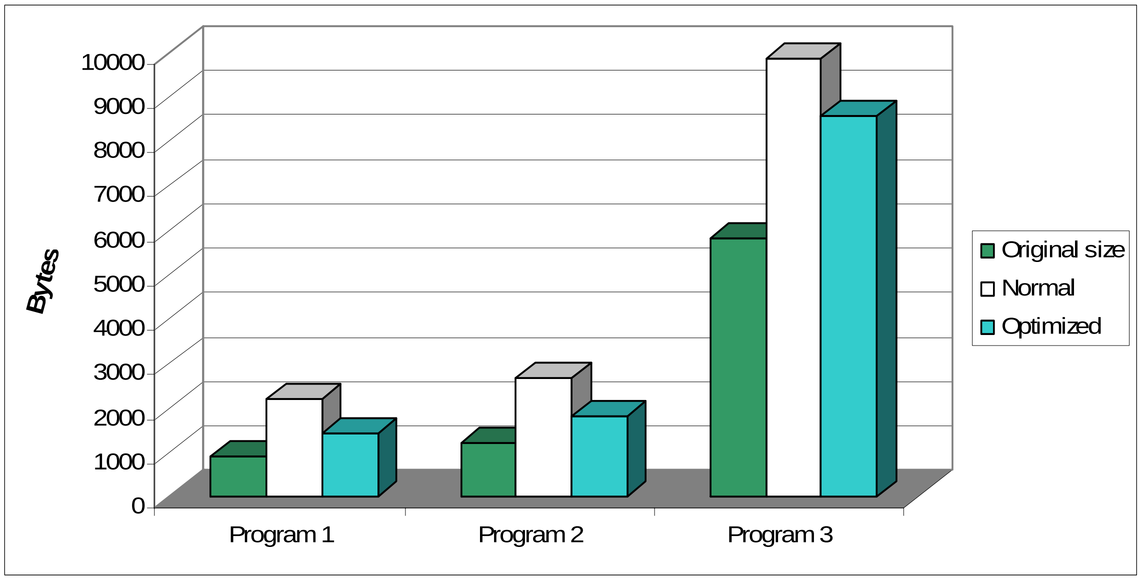


Figure 28. Code Inflation Due to the Addition of Self-healing Primitives.

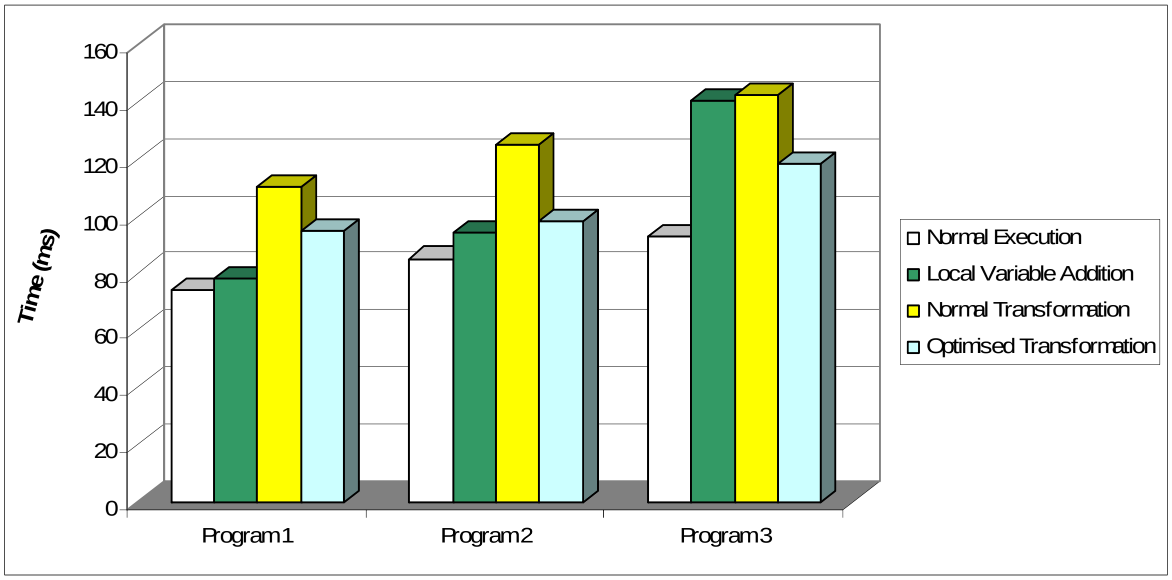


Figure 29. Execution Times of the Three Test Programs.

Figure 29 shows the timing data taken during execution. It is evident from Figure 29 that the algorithm takes linear time to execute. For the added functionalities in the code, the application programmer must be comfortable with such a small increase of pre-processing time along with the execution time. The additional memory requirement of the added data structure (Object array) is also measured. For this test, the code is instrumented in such a way that it keeps track of the amount of heap memory available to the JVM at each step of the self-healing process. On average, the runtime memory inflation is 10% to 15%. This is a minimal increase of runtime memory usage that will not typically have any significant adverse effect on the program execution.

Summary

This chapter presents techniques to add self-healing characteristics to existing object oriented programs. By working at the byte-code level, existing user code is injected with self-healing capabilities by statically analyzing the code and transforming it in such a way that the application becomes a self-healing component. The goal is to make this transition as simple and easy to the application programmer as possible in order to fulfill the goals of an autonomic system. To do that, the code is analyzed and the autonomic functionality is inserted in a manner that is separated from the service functionality of the existing code.

CHAPTER 8

CONCLUSIONS AND FUTURE WORK

Autonomic computing is a new paradigm where computing systems possess the capability of self-management. Incorporating such autonomic functionality into applications without user involvement is useful not only for application programmers who are building distributed systems from scratch, but also for users of existing systems. Although it may be desirable to build such self-adaptive distributed systems from scratch, it is not always a feasible option, mostly because of the cost and time associated with such a major development, but also because it is not practical to abandon an existing distributed or parallel application and re-program it from scratch to be self-adaptive. Programming such a distributed application is also an error-prone task and programmers need expert knowledge to handle the distribution management issues along with programming the application problem at hand. For average programmers, this becomes a daunting task when they also have to incorporate autonomic primitives into the system. In real life, programmers want to concentrate on the problem in hand, rather than spend time on incorporating autonomic behaviors in their system. It is helpful to programmers if such autonomic behaviors can be added automatically and transparently to existing systems.

This thesis presents techniques of injecting autonomic primitives into existing user code by statically analyzing the code and partitioning it to manageable autonomic components. Software architecture to provide such autonomic computing support is presented to determine its suitability for a fully fledged autonomic computing system.

The presented architecture is a novel peer-to-peer distributed object-based management automation architecture. In this model, independent or communicating objects are treated as managed elements in the geographically distributed autonomic elements. This thesis also presents a self regulating design of an autonomic element in a distributed object environment. Architectural choices have a profound effect on the capabilities of any autonomic system and affect many of the design decisions during its implementation. The goal of this architectural design is to provide an easy to program autonomic element which can be implemented in most domains with minor modifications.

As a conclusion, this research demonstrates the suitability and effectiveness of code transformation to orchestrate addition of autonomic primitives into existing applications. In a larger context, this thesis shows the feasibility of exerting externalized self-adaptation with applications that choose object oriented technology for their distributed environment and program execution. The software architecture presented in this thesis will be a precursor to the new era of self-adaptive software architecture.

This work can be pursued further in a number of directions:

- Code transformation and code injection techniques to include other autonomic properties, namely self-configuration and self-protection. Although this research injects codes for partial self-configuration, a more extensive analysis and implementation techniques are needed to fully visualize a totally autonomic system.
- Devising optimization techniques for the code transformation technique. Although the code transformation techniques presented in this thesis are optimized up to a certain extent, incorporating more code transformation for

multiple autonomic properties together might open avenues for exploring further optimization techniques.

- One of the major motivations of building large-scale self-adaptive distributed systems is to harvest idle cycles of machines in a large networked environment to get an access to unprecedented computation power. This is based on the assumption that resources are contributed by their owners to allow others to share them. However, in practice, there is no model for trust in such environments. Developing a new trust model in such a dynamic environment will certainly facilitate the implementation of autonomic systems.

REFERENCES

1. Abbas N., Palankar M., Tambe S. and Cook J. E., "Infrastructure for Making Legacy Systems Self-Managed", *2004 Workshop on Self-Managing Systems*, Newport Beach, CA, USA, October 31 - November 1, 2004, http://www.cs.nmsu.edu/please/ddl/papers/woss2004.pdf.

2. Agrawal D., Lee K. W. and Lobo J., "Policy-Based Management of Networked Computing Systems", *IEEE Communications Magazine*, Vol. 43, No. 10, pp. 69-75, 2005.

3. Ajmani S., "A Review of Software Upgrade Techniques for Distributed Systems", *Technical Report*, MIT Computer Science & Artificial Intelligence Laboratory, November, 2004.

4. Anthill, University of Bologna, Italy, http://www.cs.unibo.it/projects/anthill/, 2006.

5. Aridor Y., Factor M. and Teperman A., "cJVM: A Single System Image of a JVM on a Cluster", *International Conference on Parallel Processing (ICPP'99)*, 1999.

6. Balzer B., "Probe Run-Time Infrastructure", Dec. 2001, http://www.schafercorpballston.com/dasada/2001WinterPI/ProbeRunTimeInfrastructureDesign.ppt.

7. Bennani T., Blain L., Courtes L., Fabre J., Killijian M., Marsden E. and Taiani F., "Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization", *International Conference on Dependable Systems and Networks (DSN'04)*, pp. 549-554, 2004.

8. Birrel A. D. and Nelson B. J., "Implementing Remote Procedure Calls", *ACM Transactions on Computer Systems*, Vol. 2, pp. 33-59, 1984.

9. Bouchenak S., "Pickling Threads State in the Java System", *Third European Research Seminar on Advances in Distributed Systems*, Portugal, 1999.

10. Brown A. B., Hellerstein J., Hogstrom M., Lau T., Lightstone S., Shum P., Yost M. P., "Benchmarking Autonomic Capabilities: Promises and Pitfalls", *First International Conference on Autonomic Computing* , pp. 266-267, 2004.

11. Bruneton E., Lenglet R. and Coupaye T., "ASM: a Code Manipulation Tool to Implement Adaptable Systems", *Proceedings of the Adaptable and Extensible Component Systems Conference*, France, pp. 1-12, 2002.

12. Byte Code Engineering Library, BCEL, http://jakarta.apache.org/bcel/.

13. Cai Z., Kumar V., Cooper B., Eisenhauer G., Schwan K., Strom R. E., "Utility-Driven Availability-Management in Enterprise-Scale Information Flows", *Technical Report*, College of Computing, Georgia Institute of Technology, USA, 2006.

14. Chess D. M., Segal A., Whalley I. and White S. R., "Unity: Experiences with a Prototype Autonomic Computing System", *First International Conference on Autonomic Computing*, pp. 140-147, 2004.

15. Chiba S. and Nishizawa M., "An Easy-to-Use Toolkit for Efficient Java Byte code Translators", *2nd International Conference on Generative Programming and Component Engineering, LNCS 2830*, pp.364-376, 2003.

16. CISCO System, Adaptive Service Framework (ASF), http://www.cisco.com/application/pdf/en/us/guest/partners/partners/c644/ccmigration_09186a0080202dc7.pdf.

17. Cojen, http://cojen.sourceforge.net/.

18. Cognitive Agent Architecture (COOGER), http://cougaar.org/, 2007.

19. Constantinescu Z., "Towards an Autonomic Distributed Computing System", *14th International Workshop on Database and Expert Systems Applications*, pp. 699-703, 2003.

20. Dahm M.. "Byte Code Engineering with the BCEL API", *Technical Report B-17-98, Berlin*, 2001, http://bcel.sourceforge.net/documentation.html.

21. Danielsson P. and Hulten T., "Jdrums: Java Distributed Run-time Updating Management System", *Master's Thesis*, Department of Mathematics, Statistics and Computer Science, Vaxji University, Sweden, 2001.

22. David W., Levine W. *et al.*, "A Toolkit for Autonomic Computing", *IBM Developer Works Live*, 2005, http://www.128.ibm.com/developerworks/autonomic/overview.html.

23. Deb D. and Oudshoorn, M. J., "On Utility Driven Deployment in a Distributed Environment", *Fourth IEEE Workshop on Engineering of Autonomic Systems (EASe 2007)*, Tucson, Arizona, USA, March 26-29, 2007, Accepted for publication.

24. Deb D., Fuad M. M. and Oudshoorn M. J., "Towards Autonomic Distribution of Existing Object Oriented Programs", *International Conference on Autonomic and Autonomous Systems*, IEEE Press, California, USA, pp. 17-23, 2006.

25. Dmitriev M., “Safe Class and Data Evolution in Large and Long-Lived Java Applications”, *PhD Thesis*, University of Glasgow, May 2001.

26. Dong X., Hariri S., Xue L., Chen H., Zhang M., Pavuluri S. and Rao S., “AUTONOMIA: An Autonomic Computing Environment”, *Proceedings of the 2003 IEEE International Performance, Computing, and Communications Conference*, pp. 61-68, 2003.

27. eBiquity, University of Baltimore county, USA, http://ebiquity.umbc.edu, 2006.

28. Eclipse Development Environment, http://www.eclipse.org/, 2007.

29. Falkner Katrina E., “The Provision of Relocation Transparency through a Formalized Naming System in a Distributed Mobile Object System”, *PhD Thesis*, Department of Computer Science, Adelaide University, November 2000.

30. Feng H. T. and Lee E. A., “Incremental Checkpointing with Application to Distributed Discrete Event Simulation”, *Technical Report*, EECS Department, University of California, Berkeley, April, 2006.

31. Fuad M. M. and Oudshoorn M. J., “Transformation of Existing Programs into Autonomic and Self-healing Entities”, *14th IEEE International Conference on the Engineering of Computer Based Systems (ECBS)*, Arizona, USA, March 26 - 29, 2007, Accepted for publication.

32. Fuad M. M. and Oudshoorn M. J., “An Autonomic Architecture for Legacy Systems”, *Third IEEE International Workshop on Engineering of Autonomic Systems (EASe 06)*, Maryland, USA, April 24-28, pp. 79-88, 2006.

33. Fuad M. M. and Oudshoorn M. J., “AdJava: Automatic Distribution of Java Applications”, *Australian Computer Science Communication*, Vol. 24, No.1, pp. 65-74, 2002.

34. Fuad M. M., “Dynamic Scheduling and Load Balancing in Distributed Java Applications”, *Masters Thesis*, Department of Computer Science, University of Adelaide, 2001.

35. Ganek A. G., Hilkner C. P., Sweitzer J. W., Miller B., Hellerstein J. L., “The Response to IT Complexity: Autonomic Computing”, *Third IEEE International Symposium on (NCA'04)*, pp. 151-157, 2004.

36. Ganek A. G. and Corbi T. A., “The Dawning of the Autonomic Computing Era”, *IBM System Journal*, Vol. 42, pp. 5-18, 2003.

37. Gosling J. and McGilton H., The Java Language Environment White Paper. *Sun*

Microsystems, http://java.sun.com/docs/white/langenv/, 1996.

38. Grid Computing Information Centre, 2006, http://www.gridcomputing.com/.

39. Griffith R. and Kaiser G., "Adding Self-healing Capabilities to the Common Language Runtime", *Computer Science Technical Report*, Columbia University, 2005, http://www.cs.columbia.edu/techreports/cucs-005-05.pdf.

40. Haydarlou A. R., Overeinder B. J., and Brazier F. M. T., "A Self-Healing Approach for Object-Oriented Applications", *Proceedings of the 3rd International Workshop on Self-Adaptive and Autonomic Computing Systems*, pp. 191-195, 2005.

41. Hewlett-Packard, The Adaptive Enterprise, http://www.hp.com/products1/promos/adaptive_enterprise/us/adaptive_enterprise.html.

42. Horn P., "Autonomic Computing: IBM's Perspective on the State of Information Technology", IBM Corporation, October 15, 2001, http://www.research.ibm.com/autonomic/manifesto/autonomic_computing.pdf.

43. Hunt G. C. and Scott M. L., "The Coign Automatic Distributed Partitioning System", *Operating Systems Design and Implementation*, pp. 187–200, 1999.

44. IBM Corporation, http://www.research.ibm.com/autonomic/, 2006.

45. IBM WebSphere Software, http://www-306.ibm.com/software/websphere/, 2007.

46. Intel Corporation, Proactive Computing, http://www.intel.com/research/exploratory/.

47. Interface Definition Language, Object Management Group, http://www.omg.org/gettingstarted/omg_idl.htm, 2006.

48. Internet Inter-ORB protocol, Object Management Group, http://www.omg.org/library/iiop4.html, 2006.

49. Jarrett M. and Seviora R., "Constructing an Autonomic Computing Infrastructure Using Cougaar", *The Third IEEE International Workshop on Engineering of Autonomic Systems*, Maryland, USA, April 24-28, pp. 119-128, 2006.

50. Java Virtual Machine Specification, http://java.sun.com/docs/books/vmspec/2ndedition/ html/VMSpecTOC.doc.html.

51. Jennings N. R., "Building Complex, Distributed Systems: The Case for an Agent-based Approach", *Communications of the ACM*, Vol. 44, No. 4, pp.35–41, 2001.

52. Jennings N. R., "On Agent-based Software Engineering", *Artificial Intelligence*, Vol.

117, No. 2, pp. 277–296, 2000.

53. Julia L. L. and Gilles M., “Efficient Incremental Checkpointing of Java Programs”, *International Conference on Dependable Systems and Networks (DSN 2000)*, pp. 61-70, 2000.

54. Kaffe Virtual Machine for Java, http://www.kaffe.org/, 2006.

55. Kaiser G., Parekh J., Gross P., Valetto G., “Kinesthetics eXtreme: An External Infrastructure for Monitoring Distributed Legacy Systems”, *Fifth Annual International Workshop on Active Middleware Services*, pp. 22-27, 2003.

56. Kazman R., Yan H., Garlan D., Schmerl B. and Aldrich J., “The Recovery of Runtime Architectures”, *news@sei*, Vol. 2, pp. 1-5, 2004, http://www.sei.cmu.edu/news-at-sei/columns/the_architect/2004/2/architect-2004-2.pdf.

57. Kephart J. O., “Research Challenges of Autonomic Computing”, *The 27th International Conference on Software Engineering*, USA, pp. 15-22, 2005.

58. Kephart J. O. and Chess D.M., “The Vision of Autonomic Computing”, *IEEE Computer*, Vol. 36, No. 1, pp.41–52, 2003.

59. Landis S., “Distributed Event Notification Using RMI”, http://www.javareport.com/, July, 1999.

60. Lawall J. L. and Muller G., “Efficient Incremental Checkpointing of Java Programs”, *International Conference on Dependable Systems and Networks (DSN 2000)*, pp. 61-70, 2000.

61. Liu H. and Parashar M., “Accord: A Programming Framework for Autonomic Applications”, *IEEE Transactions on Systems, Man and Cybernetics, Special Issue on Engineering Autonomic Systems*, Editors: R. Sterritt and T. Bapty, IEEE Press, pp. 341-352, 2005.

62. Melcher B., and Mitchell B., "Towards an Autonomic Framework: Self-Configuring Network Services and Developing Autonomic Applications”, *Intel Technology Journal*, November 2004, http://developer.intel.com/ .

63. Message Passing Interface Forum, “MPI: a Message Passing Interface Standard”, http://www-unix-mcs.anl.gov/mpi/, 1994.

64. Microsoft Corporation, Component Object Model Technologies, http://www.microsoft.com/com/default.mspx, 2007.

65. Microsoft Corporation, Microsoft Dynamic Systems Initiative Overview, http://www.

microsoft.com/windowsserversystem/dsi/dsiwp.mspx.

66. MSDN developer's Manual, Microsoft Corporation, http://msdn.microsoft.com/library/default.asp, 2006.

67. Murich R., *Autonomic Computing*, IBM Press, Prentice Hall publishesr, 2004.

68. National Dysautonomia Research Foundation, General Organization of Autonomic Nervous System, http://www.ndrf.org/ans.htm.

69. NetBeans Profiler, Version 5.5, http://www.netbeans.org/products/profiler/index.html, 2007.

70. Object Management Group, Common Object Request Broker Architecture, 2006, http://www.omg.org/gettingstarted/corbafaq.htm.

71. OceanStore, University of California- Berkeley, USA, http://oceanstore.cs.berkeley.edu/, 2006.

72. Oreizy P., Medvidovic N. and Taylor R. N., "Architecture-Based Runtime Software Evolution", *20th International Conference on Software Engineering (ICSE'98)*, pp. 177 -186, 1998.

73. Orso A., Rao A., Harrold M., "A Technique for Dynamic Updating of Java Software", *18th IEEE International Conference on Software Maintenance*, pp. 649-658, 2002.

74. Oudshoorn M. J., Fuad M. M. and Deb D., "Towards an Automatic Distribution System – Issues and Challenges", *Proceedings of the International Conference on Parallel and Distributed Computing and Networks*, PDCN 2005, Innsbruck Austria, pp. 399-404, 2005.

75. Oudshoorn M. J., "Scheduling and Latency – Addressing the Bottleneck". Chapter 5 in *Architectural Issues of Web-Enabled Electronic Business* (Nansi Shi & V.K. Murthy (Eds.), Idea Group Publishing, 2003).

76. Parashar M., Liu H., Li Z., Matossian V., Schmidt C., Zhang G. and Hariri S., "AutoMate: Enabling Autonomic Grid Applications", *Cluster Computing: The Journal of Networks, Software Tools, and Applications*, Special Issue on Autonomic Computing, Kluwer Academic Publishers, Vol. 9, No. 1, 2006.

77. Parashar M., and Hariri S., "Autonomic Computing: An Overview", *UPP 2004*, Mont Saint-Michel, France, Editors: J.-P. Banâtre *et al. LNCS*, Vol. 3566, pp. 247 – 259, 2005.

78. Peyman O., Nenad M. and Richard N. T., “Architecture-Based Runtime Software Evolution”, *20th International Conference on Software Engineering (ICSE'98)*, pp. 177 -186, 1998.

79. Philippsen M. and Zenger M., “JavaParty - Transparent Remote Objects in Java”, *Concurrency: Practice and Experience*, Vol. 9, No. 11, pp. 1225–1242, 1997.

80. Project Mono, http://www.mono-project.com/Main_Page, 2006.

81. Rick K. *et al.*, “The Recovery of Runtime Architectures”, *news@sei*, Vol. 2, pp. 1-5, 2004.

82. Ritzau T. and Andersson J., "Dynamic Deployment of Java Applications", *Proceedings of Java for Embedded Systems*, London, May 2000.

83. Salehie M. and Tahvildari L., “Autonomic Computing: Emerging Trends and Open Problems”, *SIGSOFT Software Engineering Notes*, Vol. 30, No. 4. pp. 1-7, July 2005.

84. Sameer A., “A Review of Software Upgrade Techniques for Distributed Systems”, *Technical Report*, MIT Computer Science & Artificial Intelligence Laboratory, 2004.

85. Schanne M., Gelhausen T., and Tichy W. F.. “Adding Autonomic Functionality to Object-oriented Applications”, *14th International Workshop on Database and Expert Systems Applications*, pp. 725-730, 2003.

86. Schantz R. and Schmidt D. C., “Middleware for Distributed Systems”, *Encyclopedia of Computer Science and Engineering*, edited by Benjamin Wah, 2007.

87.Schmidt D. C. and Kuhns F., “An Overview of the Real-Time CORBA Specification”, *Computer*, vol. 33, no. 6, pp. 56-63, Jun., 2000.

88. Selic B, “Fault Tolerance Techniques for Distributed Systems”, *IBM developers manual*, 2004, http://www-128.ibm.com/developerworks/rational/library/114.html.

89. Segal M. E. and Frieder O., “Dynamic Program Updating: A Software Maintenance Technique for Minimizing Software Downtime”, *Journal of Software Maintenance*, Vol. 1, No. 1, pp. 59-79, 1989.

90. SERP, http://serp.sourceforge.net/.

91. Simple Object Access Protocol (SOAP), XML Protocol Working Group, World Wide Web Consortium, 2007, http://www.w3.org/2000/xp/Group/.

92. Soot: a Java Optimization Framework, Sable research group, www.sable. mcgill.ca/soot.

93. Spiegel A., "Automatic Distribution of Object-Oriented Programs", *PhD thesis*, Fachbereich Mathematik u. Informatik, Freie Universitat, Berlin, 2002.
94. Sun MicroSystems, Java Remote Method Invocation, 2006, http://java.sun.com/j2se/1.5.0/docs/guide/rmi/index.html.

95. Sun MicroSystems, Java Development Kit 5.0, http://java.sun.com/j2se/1.5.0/docs/index.html, 2006.

96. Sun MicroSystems, Java RMI over IIOP, http://java.sun.com/j2se/1.5.0/docs/guide/rmi-iiop/index.html, 2006.

97. Sun Micro Systems, Sun N1 Grid Engine, http://www.sun.com/products-n-solutions/edu/whitepapers/pdf/N1GridEngine6.pdf.

98. Tandiary J. F., Kothari S. C., Dixit A. and Anderson E. W., "Batrun: Utilizing Idle Workstations for Large-Scale Computing", *IEEE Parallel and Distributed Technology*, pp. 41-48, 1996.

99. Tanenbaum A. S. and Steen M. V., *Distributed Systems: Principles and Paradigms*, Prentice Hall Publishers, First Edition, 2002.

100. Tatsubori M., Sasaki T., Chiba S. and Itano K., "A Byte code Translator for Distributed Execution of Legacy Java Software", *Proceedings of the 15th European Conference on Object Oriented Programming (ECOOP 2001)*, Budapest, Hungary, pp. 236-255, June 18-22, 2001.

101. Teha B. *et al.*, "Implementing Simple Replication Protocols using CORBA Portable Interceptors and Java Serialization", *International Conference on Dependable Systems and Networks (DSN'04)*, pp. 549-554, 2004.

102. Tilevich E. and Smaragdakis Y., "J-Orchestra: Automatic Java Application Partitioning", *European Conference on Object-Oriented Programming (ECOOP 2002)*, Malaga, Lecture Notes In Computer Science, Vol. 2374, pp. 178-204, 2002.

103. Tobias R. and Jesper A., "Dynamic Deployment of Java Applications", *Proceedings of Java for Embedded Systems*, London, May 2000.

104. Wang M., Luo J., *et al.*, "Autonomic Element Design Based on Mind Agent Model", *International Journal of Computer Science and Network Security*, Vol. 6, No. 9B, pp. 63- 69, 2006.

105. Web Service Definition Language (WSDL), World Wide Web Consortium, 2007, http://www.w3.org/TR/wsdl.

106. White S. R., Hanson J. E., Whalley I., Chess D. M., Kephart J. O., "An Architectural Approach to Autonomic Computing", *First International Conference on Autonomic Computing (ICAC'04)*, pp. 2-9, 2004.

107. Wildstrom J., Stone P., Witchel E., Mooney R. J. and Dahlin M., "Towards Self-Configuring Hardware for Distributed Computer Systems", *Proceedings of the Second International Conference on Autonomic Computing (ICAC 05)*, pp. 241-249, 2005.

108. Wine, http://www.winehq.org/, 2006.

109. Wollrath A., Waldo J. and Riggs R., "Java Centric Distributed Computing", *IEEE Micro*, pp. 44-53, June 1997.

110. Venners B., *Inside the Java 2 Virtual Machine*, Second Edition, McGraw-Hill publishers, 2000.

111. Zhongtang C., Vibhore K. et al., "Utility-Driven Availability-Management in Enterprise-Scale Information Flows", *Technical Report*, College of Computing, Georgia Institute of Technology, USA, 2006.

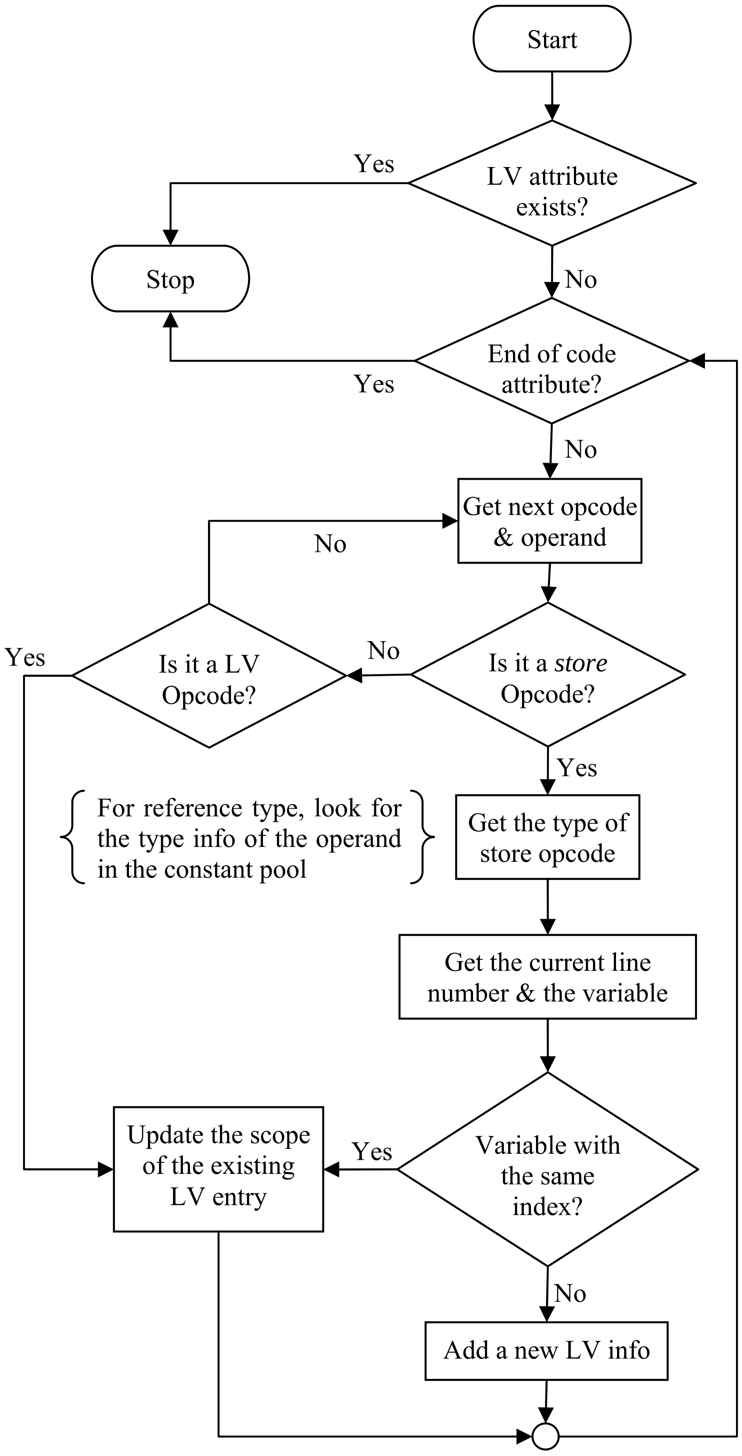
APPENDICES

APPENDIX A

FLOW CHARTS OF DIFFERENT PROCESSES

Process of Generating Local Variable Information from Byte Code

Flowchart 1 shows the process of generating local variable information from the byte code of a method. It starts first by checking whether there is already a local variable attribute attached to the method. If not, then it continues generating the attribute. To gather local variable information, the algorithm does not need to check every byte code opcode since a sub-set of byte codes only work with local variables (such as *load* and *store* operations). To optimize the algorithm, any opcode that does not work with local variables is skipped to speed up the process. The different variants of the *store* opcode indicate a write operation and hence if a *store* opcode appears with a new operand (the index into the local variable table) that indicates a new local variable that the algorithm needs to create in the local variable attribute. The type of a variable is determined by looking at the type of opcode being used to manipulate that variable. However, care must be taken not to deduce wrong types by only looking at the type of an opcode as JVM represents several data types as integers. Therefore type conversion opcodes are carefully checked to make sure of the type of a specific variable. For reference types, the opcode do not express any new information about the class of that data type. Therefore, the *constantpool* is traversed to figure out the class of a reference type variable. To generate the scope of a variable, each time an opcode of the sub-set is encountered, the corresponding local variable scope is updated.

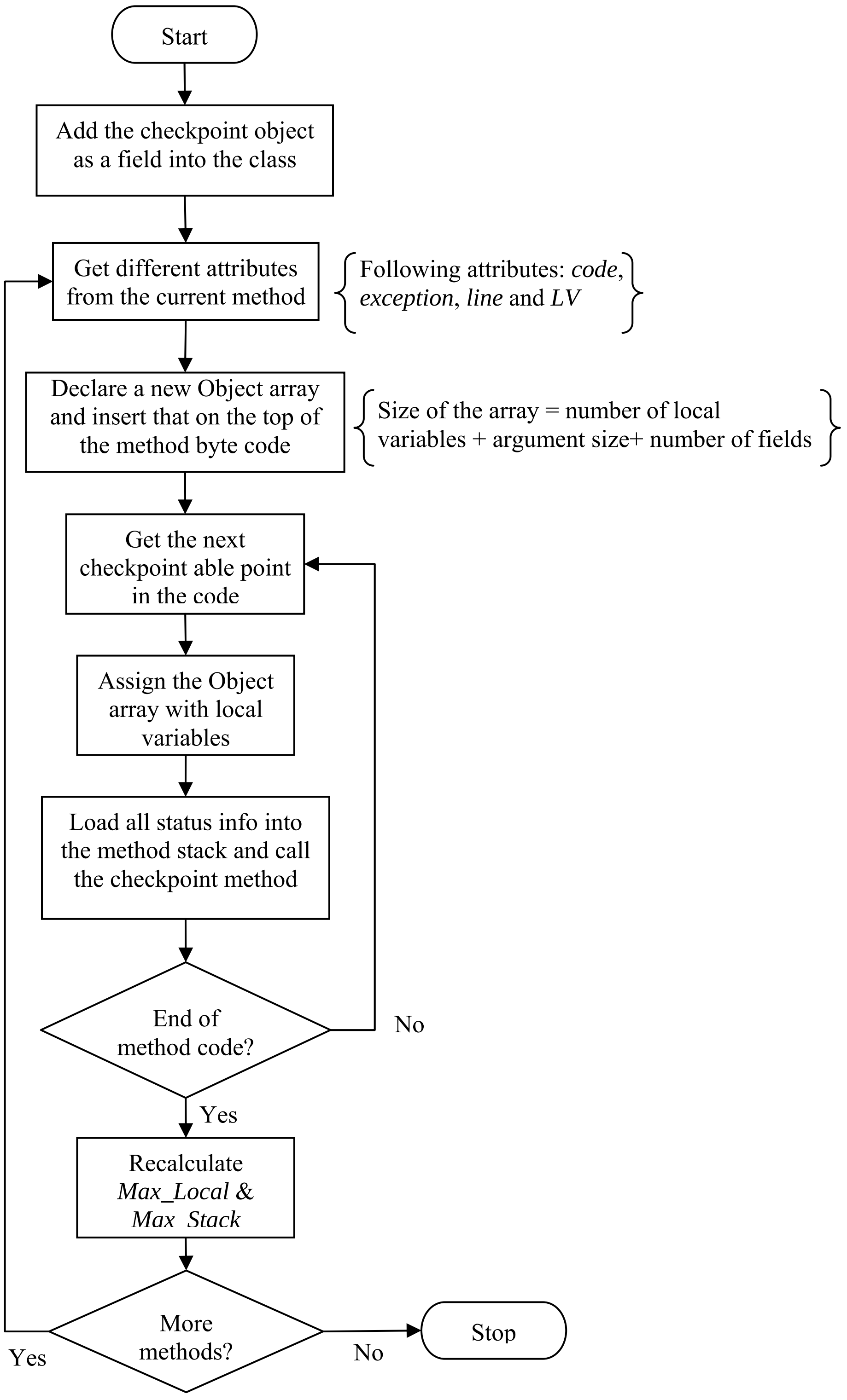


Start
LV attribute exists?
Yes
Stop
No
End of code attribute?
Yes
No
Get next opcode & operand
No
Is it a LV Opcode?
Yes
No
Is it a *store* Opcode?
Yes
For reference type, look for the type info of the operand in the constant pool
Get the type of store opcode
Get the current line number & the variable
Update the scope of the existing LV entry
Yes
Variable with the same index?
No
Add a new LV info

Flowchart 1. Generating Local Variable Information from Byte Code.

Process of Adding Self-healing Information in a Method

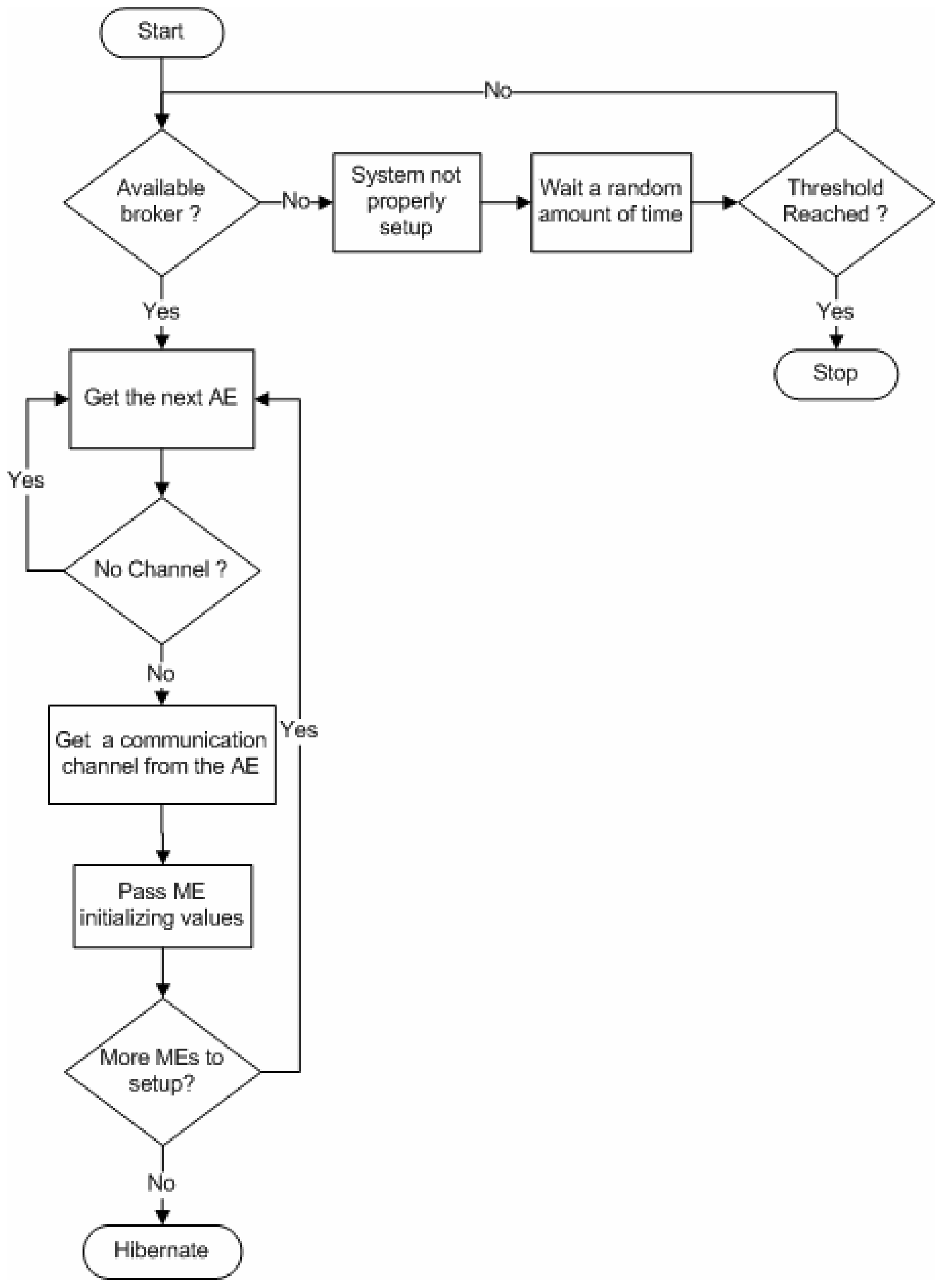
Flowchart 2 shows the process of adding self-healing primitives into a method. The assumption before running this algorithm () is that the necessary byte code attributes are already embedded in the method. Most of the attributes that are needed for this algorithm to work are by default attached to the method. Therefore the local variable attribute has to be attached with the method before running this algorithm. When assigning each local variable, fields and arguments to the Object array, proper type casting have to be performed so that during the healing process variables can be read from that array accordingly and assigned to appropriate variables. The local variable attribute is used to do appropriate type casting. A copy of the original method (and class) is backed up before making any transformations so that the user does not need to worry about restoring the original state of the method.


Start
Add the checkpoint object as a field into the class
Get different attributes from the current method
Following attributes: *code*, *exception*, *line* and *LV*
Declare a new Object array and insert that on the top of the method byte code
Size of the array = number of local variables + argument size+ number of fields
Get the next checkpoint able point in the code
Assign the Object array with local variables
Load all status info into the method stack and call the checkpoint method
End of method code?
No
Yes
Recalculate *Max_Local* & *Max_Stack*
More methods?
Yes
No
Stop

Flowchart 2. Self-healing Transformation at Byte Code Level.

Process of Setting up a Managed Element

Flowchart 3 shows the process of configuring an autonomic element (AE) with a corresponding managed element (ME). The resource repository has an broker based interface so that other autonomic elements can get a list of autonomic elements from the resource repository. If the resource repository is unreachable for any reason, the autonomic elements wait a random amount of time before attempting again. Otherwise, it asks the resource repository to give the address of the next available autonomic elements which satisfy its service requirements. Once an autonomic element is found, the two autonomic elements try to come to a service agreement. If they can, then a secure communication channel is setup between them to proceed with the managed element setup process.



Flowchart 3. Setting up an Autonomic Element with its Managed Element.

APPENDIX B

TEST PROGRAM LISTING

Test Program 1

```
import java.io.*;
import java.util.*;

public class file {

public static void main(String[] args){
    FileOutputStream fos = null;
    ObjectOutputStream oos = null;

    try {
        fos = new FileOutputStream("output.dat");
        oos = new ObjectOutputStream(fos);

        byte[] randomBytes=new byte[1024];
        oos.writeObject("My Street Address");
        oos.writeInt(59717);
        oos.writeObject(new Date());
        oos.write(randomBytes);

        fos.close();
        oos.close();

    }catch (Exception e){
        System.out.println("Exception !!!!");
        e.printStackTrace();
    }
  }
}
```

Test Program 2

```
import java.io.*;
import java.net.*;

public class sock {

    public static void main(String[] args){
        Socket aSocket = null;
        PrintWriter out = null;
        BufferedReader in = null;

        try {
            aSocket = new Socket("testServer", 7);
            out = new PrintWriter(aSocket.getOutputStream(), true);
            in = new BufferedReader(new InputStreamReader(aSocket.getInputStream()));

            out.println("A test string");
            System.out.println("echo: " + in.readLine());

            out.println(1.3999939);
            System.out.println("echo: " + in.readLine());

            char[] randomChars=new char[1024];
            randomChars=Character.toChars(124);

            out.println(randomChars);
            System.out.println("echo: " + in.readLine());

            out.close();
            in.close();
            aSocket.close();
        } catch (UnknownHostException e) {
            System.err.println("Don't know about host");
            e.printStackTrace();
        } catch (IOException e) {
            System.err.println("Couldn't get I/O ");
            e.printStackTrace();
        }
    }
}
```

Test Program 3

```
import java.io.*;
import java.util.*;

public class dmatrix {

    public static void main(String args[]){
        int nObject=10;
        int SIZE = 600;
        int numRows;
        int startRow=0;
        int endRow=-1;

        Matrix A = new Matrix(SIZE,SIZE,1);
        Matrix B = new Matrix(SIZE,SIZE,1);

        numRows=SIZE/nObject;
        multMatrix remoteMatrix[] = new multMatrix[nObject];

        for (int i=0;i<nObject;i++) {
            if (i>=SIZE%nObject){
                startRow=endRow+1;
                endRow=startRow+numRows-1;
            }else {
                startRow=endRow+1;
                endRow=startRow+numRows;
            }

            remoteMatrix[i]= new multMatrix(i,A,B,startRow,endRow);
            remoteMatrix[i].start();
            System.out.println("Started: "+i);
        }
    }
}
```

Test Program 3 (Continued)

```
class multMatrix extends Thread{

    Matrix matrix1;
    Matrix matrix2;
    private int startRow, endRow;
    private int[][] result;
    private int id,row=0;

    public multMatrix(int id, Matrix m1, Matrix m2, int sRow, int eRow){
        this.id = id;
        this.matrix1 = m1;
        this.matrix2 = m2;
        this.startRow=sRow;
        this.endRow=eRow;
        result= new int[endRow-startRow+1][matrix2.GetColumns()];
    }

    public int[][] getVal(){
        return result;
    }

    public void run(){

        for (int thisRow=startRow;thisRow<endRow;thisRow++){
            for(int col=0;col<matrix2.GetColumns();col++){
                for (int k=0;k<matrix2.GetColumns();k++)
                    result[row][col]=result[row][col]+(matrix1.GetAt(thisRow,k)*
                                                matrix2.GetAt(k,col));
            }
            row++;
        }

        System.out.println("Done --> "+id);
    }
}
```

Test Program 3 (Continued)

```
class Matrix{

    private int m_rows;
    private int m_columns;
    private int[][] m_array;

    public Matrix(int rows,int columns, int val){
        m_rows=rows;
        m_columns = columns;
        m_array = new int[m_rows][m_columns];
        InitMatrix(val);
    }

    public int GetAt(int row,int column){
        int retvalue=-1;
        if(row>=0 && row<m_rows && column>=0 && column<m_columns)
            retvalue = m_array[row][column];
        return retvalue;
    }

    public int SetAt(int row,int column,int value){
        if(row>=0 && row<m_rows && column>=0 && column<m_columns)
           m_array[row][column] = value;
        else
            return -1;

        return 0;
    }

    public int GetRows(){
        return m_rows;
    }

    public int GetColumns(){
        return m_columns;
    }

    private void InitMatrix(int value){
        for(int i=0; i<m_rows; i++)
            for(int j=0; j<m_columns; j++)
                SetAt(i,j,value);
    }

    public void DisplayMatrix(){
        for(int i=0; i<m_rows; i++){
            System.out.println();
            for(int j=0; j<m_columns; j++)
                System.out.print(GetAt(i,j) + "\t");
        }
    }

}
```