GLOBAL HOT SWAPPING IN THE SNMP SYSTEM

By

Changsong Liu, M. Sc.

A thesis submitted to
the Faculty of Graduate Studies and Research
in partial fulfilment of
the requirements for the degree of

Master of Science
in Information System Science

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
Ottawa, Canada, K1S 5B6

May 22, 2002

©2002, Changsong Liu

National Library
of Canada

Acquisitions and
Bibliographic Services

395 Wellington Street
Ottawa ON K1A 0N4
Canada

Bibliothèque nationale
du Canada

Acquisitions et
services bibliographiques

395, rue Wellington
Ottawa ON K1A 0N4
Canada

Your file Votre référence

Our file Notre référence

The author has granted a non-exclusive licence allowing the National Library of Canada to reproduce, loan, distribute or sell copies of this thesis in microform, paper or electronic formats.

The author retains ownership of the copyright in this thesis. Neither the thesis nor substantial extracts from it may be printed or otherwise reproduced without the author's permission.

L'auteur a accordé une licence non exclusive permettant à la Bibliothèque nationale du Canada de reproduire, prêter, distribuer ou vendre des copies de cette thèse sous la forme de microfiche/film, de reproduction sur papier ou sur format électronique.

L'auteur conserve la propriété du droit d'auteur qui protège cette thèse. Ni la thèse ni des extraits substantiels de celle-ci ne doivent être imprimés ou autrement reproduits sans son autorisation.

0-612-79792-9

Canada

The undersigned hereby recommend to
The Faculty of Graduate Studies and Research
acceptance of the thesis

GLOBAL HOT SWAPPING IN THE SNMP SYSTEM

submitted by

Changsong Liu, M.Sc.

In partial fulfillment of the requirements

For the degree of Master of Science

Dr. Bernard Pagurek, Thesis Supervisor

Chairman, Department of Systems and Computer Engineering

Carleton University

May 22, 2002

ABSTRACT

In this thesis, an extension of hot swapping technique, the global hot swapping, has been implemented in a distributed computing environment that dynamically updates the distributed modules in the network. No existing approaches have done the similar dynamic updating in the distributed network. The global hot swapping has been implemented in distributed environment by building a global swappable SNMP system. The swappable SNMP system allows global swapping of the SNMP engine's S-Modules in both the management station and the agents simultaneously, while the whole SNMP management system offers services continuously.

First, a method of automatically creating S-Module class and its corresponding S-Proxy class from an existing Java class is provided. And then a swappable SNMP architecture and a hot swappable SNMP system are built to implement the global hot swapping in the SNMP environment. The swappable SNMP system includes one swappable SNMP management station and several swappable SNMP agents. Two different kinds of swap managers are built according to their different functionalities. The global swap manager is built to handle the whole global swapping and the local swapping in management station; the local swap manager is built to control the local swapping in SNMP agent. A set of commands for the global hot swapping is constructed so that the whole global swapping can be dealt with by using these commands. Finally, the swappable SNMP system is tested based on a local network with 13 computers and an LQN simulation model is used to analyze the scalability of the swappable SNMP system.

ACKNOWLEDGMENTS

First I want to express my sincere gratitude to my supervisor, Professor Bernie Pagurek for his consistent guidance, enthusiasm, patience, and endless support throughout all aspects of my research. I would like to express my special thanks to Professor Babak Esfandiari and Professor C. Murray Woodside, for their invaluable help and suggestion.

I would also like to thank all colleagues who have been or are in the lab for their help. Special thanks go to Miss. Vicky Bell for her help in pre-view my thesis and my colleague Yuxiao Wang for her feedback and help.

Finally, I thank my wife, Li Connie Kuang, for her love, encouragement and support throughout all my study years. I thank my parents for their encouragement in my life. I dedicated my thesis to them.

TABLE OF CONTENTS

ABSTRACT	I
ACKNOWLEDGMENTS	II
TABLE OF CONTENTS.....	III
LIST OF FIGURES	V
CHAPTER 1. INTRODUCTION	1
1.1 BACKGROUND	1
1.2 OBJECTIVES.....	7
1.3 ORGANIZATION	8
CHAPTER 2. SOFTWARE HOT SWAPPING TECHNIQUES AND THE SNMP MODULE.....	10
2.1 HOT SWAPPING	10
2.1.1 What Is Hot Swapping?.....	10
2.1.2 Characteristics of Dynamic Software Updating.....	12
2.2 SOFTWARE HOT SWAPPING	15
2.2.1 Hot Swapping Infrastructure	15
2.2.2 S-Module.....	16
2.2.3 S-Proxy.....	18
2.2.4 Swap Manager.....	20
2.2.5 Hot Swapping Scenario	22
2.2.6 Phases of Swapping.....	24
2.2.7 S-Module and S-Proxy interfaces	25
2.3 SNMP MODULES.....	28
2.3.1 SNMP	28
2.3.2 SNMP Framework.....	30
2.3.3 SNMP Entity	31
CHAPTER 3. AUTOMATIC CREATION OF CLASSES OF S-PROXY AND S- MODULE.....	36
3.1 JAVA REFLECTION.....	36
3.2 INTERFACES OF S-MODULE AND S-PROXY	39
3.3 AUTOMATIC CREATION OF CLASSES OF S-PROXY AND S-MODULE	40
3.3.1 Class Inspector	41

3.3.2 Sprox Wizard.....	42
3.3.3 Creation of the S-Module and S-Proxy Classes.....	43
CHAPTER 4. HOT SWAPPABLE SNMP ARCHITECTURE	47
4.1 HOT SWAPPABLE SNMP	47
4.2 HOT SWAPPABLE SNMP FRAMEWORK.....	50
4.3 SWAPPABLE SNMP ENTITIES	52
4.4 SWAP MANAGERS	55
4.4.1 Global Swap Manager.....	55
4.4.2 Local Swap Manager.....	57
4.5 SWAPPABLE SNMP BASIC COMMANDS	59
4.6 THE HOT SWAPPING TRANSACTION	61
CHAPTER 5. IMPLEMENTATION AND TESTING.....	65
5.1 THE SYSTEM STRUCTURE.....	65
5.2 THE SEQUENCE DIAGRAM OF SWAPPING.....	68
5.3 HOT SWAPPING STATE DIAGRAMS	74
5.4 ISSUES CONCERNING THE NEW S-MODULE TRANSFER	79
5.4.1 Java Serialization.....	80
5.4.2 File Deployment.....	81
5.5 THE CLIENT/REQUEST STRUCTURE.....	83
5.6 TESTING	89
5.7 SCALABILITY OF SWAPPABLE SNMP SYSTEM.....	105
5.7.1 Tested Result of the Swappable SNMP System.....	106
5.7.2 LQN Model	108
5.7.3 Simulation Result.....	112
CHAPTER 6. CONCLUSIONS, CONTRIBUTIONS AND FUTURE RESEARCH	114
6.1 CONCLUSIONS	114
6.2 CONTRIBUTIONS.....	114
6.3 FUTURE RESEARCH	116
REFERENCES	117

LIST OF FIGURES

Figure 2-1. Software Hot Swapping Architecture.....	15
Figure 2-2. Referential Problem.....	19
Figure 2-3. S-Proxy for Referential Problem	20
Figure 2-4. The S-Module and its S-Proxy	25
Figure 2-5. The Evolution of SNMP	29
Figure 2-6. Traditional SNMP Manager	32
Figure 2-7. Traditional SNMP agent.....	32
Figure 4-1. A Swappable SNMP Managed Network.....	50
Figure 4-2. A Swappable SNMP Agent.....	53
Figure 4-3. A Swappable SNMP Manager.....	53
Figure 5-1. The Swappable SNMP System.....	66
Figure 5-2. The Global Swap Manager Package.....	67
Figure 5-3. The Local Swap Manager Package	67
Figure 5-4. Swapping Sequence Diagram.....	69
Figure 5-5. A Sequence Diagram showing a Failed Preparation	71
Figure 5-6. Sequence Diagram showing a Failed Swapping	73
Figure 5-7. State Diagram of Global Swap Manager.....	74
Figure 5-8. State Diagram of Local Swap Manager.....	76
Figure 5-9. State Diagram of S-Proxy.....	78
Figure 5-10. The Client/Server Structure.....	84
Figure 5-11. The Class Diagram for the Global_SendMessage.....	85
Figure 5-12. The Class Diagrams for the Local_GetMessage	87
Figure 5-13. The SNMP Manager Sending Request and Processing Response	91
Figure 5-14. The SNMP Agent Sending Response.....	92
Figure 5-15. The SNMP Manager Sending a Request after Swapping.....	94
Figure 5-16. The SNMP Agent Sending a Response after Swapping.....	94
Figure 5-17. Swapping in The Polling Process	96
Figure 5-18. The Global Swap Manager.....	98
Figure 5-19. The Local Swap Manager.....	99
Figure 5-20. SNMPv3 Message Format.....	102

Figure 5-21. The Swappable SNMP System's LQN Model	110
Figure 5-22. Comparison of Simulated and Tested Global Swapping Time	112
Figure 5-23. The Simulated Global Swapping Time	112

CHAPTER 1. INTRODUCTION

1.1 Background

Most computer programs are evolving during their lifetime [1, 2, and 3]. The range of computer program changes are wide, covering everything from fixing program bugs, improving the performance of existing system components, to modifying the functionality and structure of the computer program to adapt the old programs to new user requirements. If the program has no specific maintenance mechanism, maintenance must be done in the program's downtime. This means that the computer programs cannot provide highly available services. One approach to solving this problem is dynamic updating: the program is updated without stopping its operation. If a program were assumed to run for some relatively short period of time, dynamic updating of the program would be unnecessary. The modification could be implemented once the program has finished its current operation. The next time it runs, the updated version of the program is used. But there are many mission-critical computer programs that need to run continuously for a very long time, which means these programs must provide highly available services over a long period of time with minimized interruptions. These mission-critical systems include telecommunication systems, air traffic control systems, nuclear station control systems, and so on. Failures in telecommunication systems can result in widespread network unavailability and data lost, and failures in air traffic control systems and nuclear station control systems can even be life threatening. These highly

available computer programs must be able to be updated without their service being interrupted, or with minimized interruptions.

Dynamic updating of a computer program is an approach which changes some parts of the computer program without interrupting its execution [4, 9, 10, and 11]. The problem of dynamic updating has been analyzed for many years. The requirements of dynamic updating include robustness, scalability, flexibility, efficiency and ease of use. In general, dynamic updating is difficult to implement, because it cannot be predicted when the system is designed. This becomes even more difficult when dealing with distributed systems. Distributed computing systems are used commonly in commercial, industrial and research establishments. Usually, these systems are very complex and the software is not centralized in one place.

To meet the requirements of dynamic updating of software systems, many approaches [31, 32, 33, and 34] make use of component-based programming technologies that have many benefits. However, few provide detailed design and implementation. The main approaches that have used in component-based programming technologies are examined below.

One application, using standard component technologies, i.e. JavaBeans, was described in [8]. It provides a method of simplifying the hot swapping process. SwapBox, an extension of the BeanBox, was built as a test container for hot swappable JavaBeans. The SwapBox follows the event-based communication mechanism used in the BeanBox with

a facility to make up mapping rules at runtime, so that the state of the old bean can be mapped to the state of the new bean. Two types of swappable beans are tested: one is the `GameBoardBean`, and the other is the `SortingBean`. With the `GameBoardBean`, the new `GameBoardBean` has slight difference with regard to the old `GameBoardBean`, in that it displays colorful bars and some interface changes. With the `SortingBean`, the bubble-sorting bean is swapped out by the quick sorting bean.

A kind of runtime software evolution approach was discussed in [33]. In this approach, a kind of proxy, a connector, was used, which mediates and controls interactions among components, and thereby separates computation from communication, minimizes component interdependencies, and facilitates system understanding, analysis, and evolution. A cargo routing system was designed to test the updated user interface and architecture.

The other kind of software evolution approach, the SOFA (SOFTware Appliances) architecture, the SOFA component model and its extension called DCUP (Dynamic Component UPdating), were discussed in [34]. DCUP was based on a small set of orthogonal abstractions – Component Manager, Component Builder, Wrapper, and Updater. The SOFA component is a black box of a specified type with precisely defined interfaces. Each component consists of a permanent and a replaceable part. The utilization of SOFA in a banking system was discussed. However, DCUP was not implemented in the banking system.

The disadvantage of these existing approaches is that when they are used in dynamic updating of a distributed computing system, they cannot dynamically update those components which are distributed in the networks. They require the updated modules to be centrally hosted, mainly in the server side. But in the distributed computing environments, it is more crucial to be able to dynamically update distributed software modules. This requirement becomes even more obvious in the SNMP system. When we want to update the functionality of SNMP engines, we must update the functionality simultaneously in the SNMP management station and the SNMP agents. We want to come up with an approach, i.e. global hot swapping, which will provide a way to update distributed software modules dynamically. Global hot swapping is an extension of the hot swapping technique with the global swap manager.

The applicability of the hot swapping technique in software [4] has been considered as a solution to the problem of dynamic updating. The term “hot swapping” comes from the hardware field, and refers to the connection and disconnection of peripherals or other components without interrupting system operation. This facility has had design implications for hardware for a long time. Using hot swapping in software means replacing a software program or a part of the program while the whole software system remains in operation. The infrastructure of the hot swapping technique is discussed in [4, 5, 6, and 7]. The hot swapping technique provides a new software maintenance approach. Several approaches for designing components in the hot swapping infrastructure have been investigated and analyzed. This new concept of dynamic change has gained increasing interest.

In our software-based hot swapping infrastructure, the computer program is composed of S-Components, non-S-Module components and a Swap Manager. Each S-Component includes an S-Module and corresponding S-Proxy. The S-Modules are the only parts that can be swapped in the system. The Swap Manager controls the whole swapping process. Software hot swapping means swapping S-Modules while the whole software system remains in operation.

Using the hot swapping technique, we solve the referential transparency, state transfer, and mutual referential problems in dynamic updating of software. In our global hot swapping, the dynamically updating distributed module is the global swapping process that is constructed by several local swapping processes. These local swapping processes are distributed in the network. The global swap manager handles the transaction of global swapping process and the local swap manager handles the transaction of local swapping process. Using the global swap manager, we can handle the distributed transaction of the global hot swapping.

The global swapping is more complex than the local swapping. The hot swapping technique has been tried in some programs [6, 7]. In [6], Ning built a swappable SNMP agent. The swapping happened only in the SNMP agent, so the swapping is only a local swapping with no reference to distributed hot swapping in a distributed computing environment. In [7], an experimental application was built to show how to swap different S-Modules. These experiments focused on local swapping, i.e. swapping which occurs

only in one location, but did not make use of hot swapping techniques in the distributed environment, where global swapping takes place in several different locations. When using hot swapping in a distributed environment, we must make sure the global swapping is consistent. We chose the SNMP network management system to test the global hot swapping.

Since SNMP [12] was developed in 1988, the Simple Network Management Protocol has become the de facto standard for network management. SNMP is extensible so that different vendors can add network management functions easily to their existing products. SNMP broadens the base of multi-vendor support by separating the management architecture from the architecture of the hardware devices. In the SNMP network management system, there are two primary elements: a manager and agents. The manager is the console through which the network administrator performs network management functions. Agents are the interfaces to the actual device being managed. The network administrator can get information about the whole network through the manager and agents. Bridges, hubs, routers or network servers are examples of managed devices that contain managed objects. We chose the SNMP network management system and wanted to test global hot swapping by dynamically upgrading the SNMP engine's functions in both the manager and the agents simultaneously while the whole management network continuously provides service. The global hot swapping should act atomically: either the whole swapping is successful so that the system uses the new S-Modules, or the whole swapping rolls back so that the system uses the original S-Modules.

1.2 Objectives

In this thesis, the automatic creation of the classes of S-Proxy and S-Module will be discussed first. In our hot swapping system, every hot swappable application must have S-Components that are constructed by S-Modules and corresponding S-Proxy components. For brand-new hot swappable applications, both the S-Module and S-Proxy will be written from the ground up. But for existing applications which need to be swappable, there should be a way to modify the existing application easily without having to rewrite every module manually. The method of creating the corresponding classes of S-Proxy and S-Module from the existing class is discussed in more detail in Chapter 3. We used Java reflection to automatically create the S-Proxy and S-Module classes from the original module.

The major goal of this thesis is to use global swapping build a global hot swappable system for SNMP that could use global swapping in the SNMP system. The hot swappable SNMP architecture was discussed and a hot swappable SNMP system was built to implement the global hot swapping process in the distributed environment. This hot swappable SNMP system was built based on Ning's swappable SNMP agent. In the hot swappable SNMP system, the global swapping will happen in both the manager and the agents simultaneously. The global swapping must keep the swapping transaction's atomicity. The hot swappable SNMP system contains one swappable SNMP management station and several swappable SNMP agents. Both the SNMP manager and

agents include swappable SNMP engines built based on the S-Module. Two different kinds of swap managers were built according to their different functionalities. The global swap manager was built to handle the whole global swapping process and the local swapping process in SNMP management station. The local swap manager was built to handle the local swapping process in the SNMP agent. The functionality of the global swap manager and the local swap manager were analyzed in detail. A set of commands for the global hot swapping was built so that the whole process of swapping can be handled by using these commands. A simulation model was built to analyze the scalability of the hot swappable SNMP system. The simulation result showed that the global swapping time grows linearly with the increase of swappable SNMP agents and the network delay.

Using the constructed hot swappable SNMP system, we implemented a flexible and dynamically extensible environment for the experiments with the hot swapping program in distributed computing, determine how the global swapping can be handled in a distributed computing environment, and estimated the global swapping time in swappable SNMP system.

1.3 Organization

The remainder of the thesis is organized as follows.

Chapter 2 briefly overviews the current hot swapping techniques and the history and modularity of SNMP.

Chapter 3 discusses how to automatically generate the S-Proxy and S-Module classes from an existing Java class. Java Reflection is mainly used for this automatic generation.

Chapter 4 presents the reasons for using the hot swapping technique in the SNMP network and how to control global hot swapping in the SNMP network management environment. The hot swappable SNMP architecture and the swapping transaction are discussed, and a hot swappable SNMP system is built. This hot swappable SNMP system supports the dynamic swapping of different S-Modules while always providing continuous network management services.

Chapter 5 focuses on implementation and application issues related to the hot swappable SNMP system. The sequence diagrams of swapping, the state charts of different components, and the modeling of swappable SNMP system are discussed. The constructed hot swappable SNMP system is implemented and tested. The scalability of the hot swappable SNMP system is discussed using a simulation model (LQN model).

Chapter 6 provides the conclusion of this thesis. The contributions and recommendations for future research are also discussed.

CHAPTER 2. SOFTWARE HOT SWAPPING TECHNIQUES AND THE SNMP MODULE

The dynamic updating of a computer program is the approach which changes some parts of the computer program without interrupting the computer program's execution. The applicability of the hot swapping technique [4] has been considered as a solution to the dynamic update problem. In this chapter, we will set out the architecture of hot swapping and the SNMP framework.

2.1 Hot Swapping

2.1.1 What Is Hot Swapping?

The term "Hot Swapping", as used in the hardware field, refers to the changing and upgrading of peripherals or other components during runtime without interrupting system operation. Hot swapping is a seductive concept: find the component which needs to be changed, pull it out, and swap a new component in its place without shutting down the whole system. This facility may have design implications for both hardware and software. There is an increasing recognition of the need for support for software-based hot swapping. Software hot swapping means replacing a software program or part of a program while the whole software system remains in operation.

The reasons to allow hot swapping are obvious. Most software systems are evolving during their lifetime. The range of software change is wide, covering everything from

bug fixing of existing system components, to performance improvements, to changes of the functionality and structure of the system necessary to adapt the system to new user requirements. If the program does not have a specific mechanism for maintenance, this maintenance has to be done in the program's downtime. This means that the computer programs cannot provide highly available services. But systems like telecommunication systems, air traffic control systems and nuclear station control systems cannot afford any downtime. The failure of telecommunication systems can result in widespread network unavailability and data lost. Failures in air traffic control systems and nuclear station control systems can even threaten lives. For these kinds of mission-critical systems, a hot swapping technique is required to assure that these systems are highly available and that the loss of system functionality during updating is minimized. So the hot swapping technique forms the backbone of high availability systems.

There are differences between the hot swapping technique and dynamic linking. When we discuss dynamic linking we will also discuss dynamic binding, since these two techniques are very easy to confuse, although they are totally different things. Dynamic binding means the mapping identifier to name is done at run time so that the code executed to perform a given operation is determined at run time from the class of the operand. It is one of the properties of object-oriented programming languages. Dynamic linking is the dynamic linking of program libraries in the runtime environment. Many modern programming languages provide dynamic linking. For example, C, C++ and Java use facilities such as the COM, DCOM API or the Java ClassLoader class to provide dynamic linking. Dynamic linking means linking software modules to client programs

dynamically at execution time. But hot swapping means that we can swap modules when they are running. In dynamic linking, the modules can be linked at execution time, but we cannot change the linking to another module while the current module is running. The ideal hot swapping is that when one module is running, the module is blocked at a checkpoint. The active state of this module, including all local variables and stacks of this module, is captured and mapped to a new module so that the new module will begin running from where the original module is blocked. Most importantly, during this swapping, the whole software system remains in operation.

2.1.2 Characteristics of Dynamic Software Updating

The techniques a system uses for dynamic updating are influenced by the application domain and the desired performance and correctness guarantees. For example, a system for dynamically updating an information server used in the time-sharing environment would generally not be appropriate for dynamically updating a real-time process system. But there are some characteristics, as documented by Segal [11], which most dynamic update techniques must satisfy, regardless of their intended uses. These characteristics for hot swapping are discussed below:

- Program robustness. Many applications needing the hot swapping technique are mission critical: they must continuously provide correct service. Dynamic updating must provide program correctness so that the updated system runs just as well as at times when no updates are in progress.

-
- **Swapping safety.** The hot swapping must be safe. In particular, a safe hot swapping will not perform illegal operations that lead the system to crash. Typically, notions of safety encompass type-safety, which is a well-understood program language concept. The Java Virtual Machine has popularized the use of safety as a security mechanism for loaded code.
 - **No constraint on swapping time.** Usually, when the hot swapping occurs is very important. Choosing the wrong time may result in inconsistent states due to race conditions. The ideal hot swapping system should show that the timing of a swapping would not result in an error.
 - **Support low-level program change.** To dynamically update a range of programs, a hot swapping system must support a variety of low-level program changes. The simplest kind of change is to replace a module with a new one that is implemented differently. More complicated changes include changing the module's interface, having the module retain states between invocations, changing the state's implementation, and changing the implementation of state variables, which often occurs in programs that implement data structures as abstract data types.
 - **Support code restructuring.** Significant code restructuring can occur during maintenance - a change beyond simple module replacement. The hot swapping

system must be able to update programs when new modules are added, existing modules are deleted, and functionality is moved between modules.

- Available in distributed environment. Many programs that benefit from dynamic updating are distributed by nature. The hot swapping system must scale to large distributed programs, in which there are hundreds of thousands of individual modules, and cooperate with other updating systems across administrative domains.
- Do not require special-purpose hardware. Usually the hot swapping system should not need spare hardware to be added to the system.
- No constraint on language and environment. The user must be free to choose a language and system environment. The hot swapping system must not force programmers to write code or call operating system primitives in a radically different manner. Ideally, updating systems should tolerate a variety of programming styles and allow substantial code reorganization between versions of the program being updated.

Bearing in mind these characteristics that the hot swapping system should satisfy, the next section will discuss the details of software hot swapping systems.

2.2 Software Hot Swapping

2.2.1 Hot Swapping Infrastructure

As shown in Figure 2-1, in the hot swapping architecture [4, 5, 6, and 7], an S-Application is composed of S-Components that are each made up of an S-Module and a corresponding S-Proxy, non-S-Module components, and a Swap Manager.

Figure 2-1. Software Hot Swapping Architecture

Under this construction, only the S-Module can be swapped at run time. Both the S-Module and the non-S-Module can interact with other S-Modules, non-S-Module components, and clients. The clients in the hot swapping system may be external, demanding services from the application, or internal, meaning the interaction of one S-Application module with another. But only a non-S-Module can have a direct interface to provide its services. The use of the S-Module requires going through the S-Module's S-Proxy. The S-Proxy also controls the S-Module's communication with the Swap

Manager. The main difference between an S-Module and a non-S-Module is that the upgrading of an S-Module will not interrupt the running of the S-Application, while upgrading a non-S-Module generally requires re-compiling, re-linking and re-starting the S-Application. The application has to be configured into a swappable S-Module in order to be modified by swapping the S-Module, while keeping the whole application's concurrence. The application can be composed of multiple S-Modules. There is only one Swap Manager for an application. The Swap Manager controls the transaction of swapping the S-modules and has access to all S-Modules. The Swap Manager has no control over those non-S-Modules.

2.2.2 S-Module

Since the S-Module is the only part that can be swapped in the hot swapping system, the S-Module must have some special characteristics [4]. These characteristics are discussed in [4, 5, 6, and 7], as follows:

- **Identity:** An S-Module must have a unique identity. In the hot swapping application, the Swap Manager controls all the S-Modules in the application. When the S-Module is instantiated, it must register with the Swap Manager so that the Swap Manager can access and swap the S-Modules using this identity.
- **Version:** An S-module has an associated version with major and minor indices and annotations that describe the nature of the changes from a previous version. In fact,

the version is a very important part of identity. The different version indicates the different S-Modules.

- **Service:** An S-Module provides a set of services that show the behavior or functionality of the S-Module. There are two types of services provided by an S-Module. One type is the behavior service, which specifies the functions that the S-Module can provide at the request of its clients. Another type of service is the swapping service, which is a framework-defined service. The swapping service specifies the functions that relate to swapping. Through the swapping services, the Swap Manager can ascertain the internal states of the S-Module, set the internal states of the S-Module, check whether the S-Module is in a swappable state, etc.
- **Internal state:** The internal state of the S-Module summarizes its attribute values and its execution status. It is very important since it affects the process of hot swapping. In some states the S-Module is swappable, in others it is not. If and only if an S-Module is in a swappable state, the hot-swapping transaction can take place. If the swapping transaction takes place, the original S-Module's internal state should be mapped to the new S-Module and the new S-Module should provide services based on the internal state of the original S-Module. If the swap transaction is aborted, the original S-Module should continue its service based on its previous state.

- **Dependency list:** Any S-Module should have a dependency list that includes the S-Modules that it depends on. Before an S-Module can be instantiated, all the S-Modules that it depends on must have been loaded into the S-Application.
- **Mapping rules:** In the hot swapping system, there exists a set of mapping rules. These mapping rules control the transfer of the original S-Module's internal state to the new S-Module.
- **Persistence:** During the hot swapping transaction, the system resources held by the original S-module should be released or transferred to the new S-Module. The new S-Module should continue the execution of the original S-Module from where the original S-Module is swapped.

2.2.3 S-Proxy

In the hot swapping system, using the S-Proxy, the S-Module's referential problem is solved successfully. The referential problem [4, 7] is caused by the S-Module's interaction with the other parts of the hot swapping system. The following example explains the referential problem.

Figure 2-2. Referential Problem

In Figure 2-2(a), object A requests the S-Module S for service. If we want to use a new S-Module S' to replace the S-Module S, object A has to change the reference from S to S'. This means that the object A must know about the change. However, as depicted in Figure 2-2 (b), if the object A has passed S's handle to another object B, then B will retain the reference to S even if A has changed the reference to S', unless object A informs object B the change. Thus, if we want to swap S-Module S with the new S-Module S', we have to inform all the objects that have the S-Module S's handle to switch the reference from S to new S-Module S'. In some cases, who has the S-Module's handle is unknown, and thus it is difficult to notify all objects to switch the handle to the new S-Module.

By using the S-Proxy, the S-Proxy controls the access to the S-Module so that the S-Proxy can hide the real handle of the S-Module from those clients. Instead of giving out the S-Module's handle, the S-Proxy's handle is given out to the clients.

Figure 2-3. S-Proxy for Referential Problem

In Figure 2-3, when the S-Module S is swapped, the S-Proxy will change the handle from S to the new S-module S'. The clients retain their relationship with the S-Proxy and know nothing about the swapping. This makes the swapping transparent to those clients. All service requests sent to the original S-module are automatically switched to the new S-Module through the S-proxy.

2.2.4 Swap Manager

The Swap Manager is the core of the hot swapping architecture. It controls all the S-Modules and the transaction of swapping the S-Module. In the distributed hot swapping system there are different kinds of swap managers. Usually there should be a global Swap Manager that handles the whole distributed swapping process and local Swap Managers that handle the local swapping processes. The services that these different Swap Managers handle are discussed in detail in Chapter 4. The general services that all Swap Managers must provide are discussed below:

- **Listening services:** The Swap Manager provides a listening service to receive the message about the creation of new S-Modules and initialization of S-Modules.

-
- **Security services:** Security is a very important issue in hot swapping systems. When the Swap Manager receives the swapping message, it has to be sure of the following issues:
 - The confidentiality of data
 - Authentication of the source
 - Integrity of the data
 - **Timing service:** Timing is critical in the swapping process. The Swap Manager must control how long the swapping transaction can take, when the swapping can begin, and when to block the application services. The time constraints are particular to different systems.
 - **Transaction service:** The Swap Manager must provide control of the hot swapping transaction. The transaction decides whether the swapping process will succeed or not. If the swapping fails, the original S-Module can rollback and continue to provide services.
 - **Naming Service:** The Swap Manager must provide the naming service for the S-Module so that the S-Module will be registered and can be identified using the S-Module's identity.

- **Repository:** The Swap Manager must provide the caches for the state of the S-Module during the swapping transaction.
- **Garbage collection:** The Swap Manager must provide garbage collection so that after the swapping the original S-Module will be destroyed and the space can be released.

2.2.5 Hot Swapping Scenario

In a hot swapping system, an S-Module and its S-Proxy constitute an S-Component. The hot swapping scenario is as follows:

1. The Swap Manager gets a message from its listening service. The incoming message includes all the information about the new S-Module, such as identity, the version of the new S-module, the code for instantiating objects, etc.
2. The Swap Manager will check the new S-Module's security and authentication. If successful, the new S-module will be instantiated.
3. The Swap Manager will search the S-Proxy of the original S-Module, and send a preparing swapping message to the S-proxy so that the S-proxy will block any new method calls and start the timing service.

-
4. The Swap Manager will then check with the S-Proxy whether the original S-Module is in a swappable state. If it is not in a swappable state, the Swap Manager will wait until the S-Module finishes the interaction and enters swappable state.
 5. When the original S-Module is in a swappable state, the Swap Manager will call the S-Proxy to get the internal state of the original S-Module and map the state to the state of the new S-Module. The mapping rules are applied here.
 6. After the new S-Module has been successfully initialized, the Swap Manager initiates a hot-swapping transaction by changing the reference to the old S-Module in the S-Proxy with the reference to the new S-Module.
 7. The timing service stops timing and checks to see if the transaction is finished. If yes, this hot swapping transaction is successful. The Swap Manager tells the S-Proxy to release the blocked calls. The original S-Module can now be removed.
 8. If, during step 4 to step 6, the timing service gets a time-out event, this transaction fails. The Swap Manager sends out a notification about this transaction and releases the blocked calls.

2.2.6 Phases of Swapping

This section provides a more detailed description of the swapping process. Usually the swapping process is subdivided into four phases: preparation, mapping, swapping, and finishing.

Preparation phase: this is the period from the initiation of the swapping to the beginning of the mapping. Before an S-Module is swapped, it has to make sure that the S-Module is in a swappable state. The swap manager asks the S-Proxy to block the original S-Module and capture the state of the original S-Module. Since the S-Module is blocked, the swappable state will be unchanged during the swapping process.

Mapping phase: in this phase, the new S-Module is created and initialized using the original S-Module's state. This mapping must ensure that the new S-Module will execute successfully and the swapping will not interrupt the execution of the whole system.

Swapping phase: in this phase, the system has to be reconstructed. The new S-Module now begins running and the S-Proxy switches the handles to the new S-Module. All handles to the original S-Module have been cut off. This is like hardware swapping, where the new part is plugged in and the old part is taken out.

Finishing phase: in this phase, the block is released and the new S-Module begin sprocessing requests from clients. Since the S-proxy controls the access to the S-Module,

the clients do not know about the swap. Now the original S-Module will be isolated and the Swap Manager will destroy it.

From the description of these four phases, we know that whether an S-Module can be swapped depends on the status of its state and how the state is reached. But by now there is nothing to make sure whether the original S-Module is in a swappable state or not when the Swap Manager asks for the swapping process to begin.

2.2.7 S-Module and S-Proxy interfaces

Figure 2-4. The S-Module and its S-Proxy

As described in [4, 5, 6 and 7], an S-Module must implement the following interfaces, among which only the BehaviorInterface corresponds to the services provided by the originating module:

- **SwapInterface:** This interface allows the S-Module to participate in a swap transaction, preparation for swapping, voting and cleanup upon transaction commit.
- **IdentityInterface:** This interface allows the S-module to be identified. Each S-module has a unique identity so that the S-Manager can identify it.
- **BehaviorInterface:** This interface shows the actual behavior of an S-Module and allows its S-Proxy to use services provided by this S-Module.
- **MappingStateInterface:** This interface maps the states of identical S-Modules from one version to another. The Swap Manager uses this interface to properly initialize a new S-Module.
- **RetrieveStateInterface:** This interface retrieves the states of the current S-Module, or it could return the states of any of the previous versions.
- **DoesNotUnderstandInterface:** This interface handles exceptions when a service is requested but it is not provided by this S-Module.

Also, the corresponding S-Proxy must implement the following interfaces:

- **BehaviorInterface:** This interface publishes the behavior and services of the underlying S-Module. The implementation forwards external calls to the corresponding S-Module.
- **NewBehaviorInterface:** This interface is used by external clients to send dynamic messages to the methods that do not exist in the S-Proxy's Behavior-Interface.
- **SignalInterface:** This interface is used by the Swap Manager to check and control access to the S-Proxy.
- **RetrieveStatesInterface:** This interface is used by the Swap Manager to retrieve the states of the current corresponding S-Modules during a hot-swapping transaction. The request is passed through to the associated S-Module.
- **HandleCallExceptionInterface:** This interface allows the designer of an S-Component to handle the exceptional cases that may happen inside the S-Proxy.
- **SwapControlInterface:** The `SwapControlInterface` extends the `SwapInterface`. Through this interface the Swap Manager controls the transaction of the hot swapping procedure. Security is applied here to ensure that only the Swap Manager uses the interface.

2.3 SNMP Modules

2.3.1 SNMP

The Simple Network Management Protocol (SNMP) [12, 13, and 14] is a set of standards for network management, including a protocol, a data base structure specification, and a set of data objects. SNMP was designed to be an application-level protocol that is part of the TCP/IP protocol suite. The manager process achieves network management by using SNMP. Between the manager and managed devices, there exists a client/server relationship. The client program (called the network manager) makes a call to a server program (called the SNMP agent) which executes on a remote network device to get the information about the status of the device. The database, controlled by the SNMP agent, is referred to as the SNMP Management Information Base (MIB), and is a set of standard statistical and control values. SNMP allows the extension of these standard values as well as values specific to a particular agent through the use of private MIBs.

There are different versions of SNMP (v1, v2, and v3). The evolution of SNMP is displayed in Figure 2-5. SNMPv1 became both an open IETF standard and a de facto industry standard due to wide market acceptance since 1989. A broad range of vendors implemented SNMP versions and extended the scope of SNMP in many directions, including network management, system management, application management, manager to manager communication, and proxy management of legacy systems. The capabilities of SNMPv1 are fine for implementing a basic network-management system. To enhance

this basic functionality, the IETF working group that developed SNMPv2 wanted to include security functionality in the new version. Various proposals were put forward (SNMPv2c, SNMPv2u, SNMPv2*) but none of those were adopted as an IETF standard. The working group was not able to reach an agreement on how to define the required security mechanism.

Figure 2-5. The Evolution of SNMP

The third version of SNMP is derived from and built upon both the first and the second versions. SNMPv3 [14, 15, 16 and 17] allows new capability for open, interoperable, and secure management in the Internet environment. It supplements the SNMPv2 framework by supporting the following functionalities: security (methods for authentication, encryption, privacy, authorization, and access control) and an administrative framework (naming of entities, user names and key management, notification destinations, proxy

relationships, remotely configurable via SNMP operations). With SNMPv3, a single protocol entity may provide simultaneous support for multiple security models, as well as multiple authentication and privacy protocols.

2.3.2 SNMP Framework

The SNMP framework emphasizes the use of modularity for the evolution of portions of SNMP so that the general framework doesn't need to be changed. The objective of the SNMP framework is to allow evolution with richer and more clearly defined components but keeping the fundamental architecture consistent. The SNMP framework emphasizes the use of modularity for the evolution of a portion of the SNMP, without requiring a redesign of the general framework.

The objective of the SNMPv3 framework is to allow for the evolution of the implementation over a wide range of operational environments. To achieve this goal, the SNMPv3 uses a modular architecture and reuses the existing work in the previous versions as much as possible, since this existing work has implementation experience. As a result, the SNMPv3 security features rely heavily on SNMP v2u and SNMPv2* to address the need for security support and make it possible to evolve the framework to support different security models, different features and different configurations. The framework of SNMPv3 allows the different modules to be reused independently of the version of the SNMP protocol. As defined by the [13, 16], the framework design used encapsulation, cohesion, hierarchical rules, and loose coupling to reduce the complexity

of design and make the evolution of portions of the framework possible. Encapsulation describes the practice of providing a well-defined interface to a set of functions in a way that hides the details that are used internally in these functions. Cohesion describes the grouping of similar functions so that their differences are ignored and their common parts are clearly displayed. By cohesion, the similar functions can be dealt with as a single entity. It is important that the functions that are grouped together are actually similar. Functionality can be grouped into hierarchies where each element in the hierarchy receives general characteristics from its direct superior, and passes on those characteristics to each of its direct subordinates. Coupling describes the amount of interdependence between parts of a system.

2.3.3 SNMP Entity

The SNMP entity defined by [14] is composed of an SNMP engine and one or more applications. The SNMP engine provides services for sending and receiving messages, and controlling access to the managed objects. An SNMP engine contains a Dispatcher, a Message Processing Subsystem, a Security Subsystem and an Access Control Subsystem. Figure 2-6 and Figure 2-7 show the SNMP manager and the SNMP agent that include the SNMP entity, which is composed of applications such as command generators, notification originators and receivers, and the different modules of the SNMP engine (Dispatcher, Message Processing Subsystem, and Security Subsystem).

Figure 2-6. Traditional SNMP Manager

Figure 2-7. Traditional SNMP agent

The Dispatcher allows for concurrent support of multiple versions of SNMP messages. The dispatcher sends and receives messages to and from the network, determines the version of an SNMP message, interacts with the corresponding Message Processing Model, and provides an abstract interface to SNMP applications for delivery of a PDU. There is only one Dispatcher in an SNMP engine.

The Message Processing Subsystem is responsible for preparing messages for sending and extracting data from received messages. It contains multiple Message Processing Models correspond to the different SNMP versions (SNMPv3, SNMPv1 and SNMPv2 Message Processing Model) or to other Message Processing Models.

The Security Subsystem provides services such as the authentication and privacy of messages and potentially contains multiple Security Models (User-Based Security models or other Security models). A security model defines the threats that should be prevented, the service that should be provided and the security protocols that will be used (procedure and MIB data) to accomplish services such as authentication and privacy.

The User-Based Security Model (USM) for SNMPv3 defines the elements of the procedure for providing SNMP message-level security. USM protects against the following primary and secondary threats: modification of information, masquerade, message stream modification, and disclosure. The USM uses MD5 and the Secure Hash Algorithm as key hashing algorithms for digest computation. This provides data integrity

to directly protect against data modification attacks, to indirectly provide data origin authentication, and to defend against modification attacks. The USM uses the Data Encryption Standard (DES) in the cipher block-chaining mode (CBC) to protect against disclosure. The configuration parameters in the MIB (including key distribution and key management) can be remotely monitored and managed. A single protocol entity may provide simultaneous support for multiple security models, as well as multiple authentication and privacy protocols. All of the protocols used by the USM are based on symmetric cryptography, i.e. private key mechanisms. The SNMPv3 architecture permits the use of public key cryptography.

The Access Control Subsystem provides authorization services by means of one or more Access Control Models. An Access Control Model defines a particular access function to support decisions concerning access rights. An example of an Access Control Model is the View-Based Access Control Model. The View-Based Access Control Model defines the elements of the procedure for controlling access to management information. It includes an MIB for remotely managing the configuration parameters for the View-Based Access Control Model. The View-Based Access Control Model can simultaneously be associated with multiple Message Processing Models and multiple Security Models in a single-engine implementation.

The advantage of the SNMP framework is its very modular implementation. This modularity makes it easy to identify several modules as S-Module candidates for hot swapping. In [6], the swappable SNMP agent was built using the hot swapping technique

on a working Modular SNMPv3 engine that was developed by the Laboratoire de téléinformatique de l'Université du Québec à Montréal [19]. In this thesis, we built a global swappable SNMP system using these swappable SNMP agents, such that the SNMP entities can be dynamically changed during run time. The distributed hot swapping will happen in both the SNMP manager and SNMP agents and act atomically: either all swapping is successful, so that both the SNMP manager and SNMP agents will use the new S-Module, or all swapping rolls back so that both the SNMP manager and the SNMP agents will use the original S-Modules. During swapping, the management station will continue polling to collect the data from agents so that the management function will not be interrupted.

CHAPTER 3. AUTOMATIC CREATION OF CLASSES OF S-PROXY AND S-MODULE

As described in [5], in the hot swapping system, the S-Components are constructed by the S-Module and the corresponding S-Proxy and only the S-Modules can be swapped. For brand-new hot swappable applications, both the S-Module and the S-Proxy classes will be written from scratch. But for some existing applications that need to become swappable, the modules that need to be swapped have to be changed into the S-Modules. This means that their classes must satisfy the special requirements of the S-Module so that these modules can become S-Modules. There should be a way for these existing classes to be modified easily without having to manually rewrite them. The ideal is that for every module that needs to be swapped, corresponding S-Module and S-Proxy classes must be created. In this section, we discuss how to make the S-Module and S-Proxy classes from the original module class through the use of Java reflection.

3.1 Java Reflection

Package `java.lang.reflection` [22] provides a way to analyze the Java class files. The reflection package allows the JVM to answer questions about the classes it has loaded and the classes which do not have any existing instances yet. With the reflection API, we can determine the class of an object; get information about a class's modifiers, fields,

methods, constructors, and superclasses; find out what constants and method declarations belong to an interface, and so on.

In our application, we will create the S-Module and S-Proxy by using instances of the classes Field, Method, and Constructor obtained through the methods `getDeclaredField`, `getDeclaredMethod`, `getDeclaredConstructor`, `getDeclaredFields`, `getDeclaredMethods`, and `getDeclaredConstructors` of class “Class”. Definitions of these classes and methods are listed below.

- **Field class:** provides information about, and dynamic access to, a single field of a class or an interface. The reflected field may be a class (static) field or an instance field.
- **Method class:** is a representation of a single method, with a particular name and list of parameters (taken together, the name and parameter list form the signature of the method). A method also has a return type, a list of modifiers (described in the next paragraph), and a list of exceptions that the method might throw. We can even execute the method represented by a Method object by calling its `invoke()` method.
- **Modifier class:** represents a modifier of a method or a field in a class. Modifiers indicate some property of a method, field, or class. For example, they can indicate whether a field is static, transient, or both; whether a class is abstract or is actually an interface; or whether a method is synchronized.

- **Constructor class:** is similar to class Method, but it represents a constructor instead of a method. It has no method for getting the return value class, since constructors don't have return values.

The above was an indication of classes that represent the parts of a class (its methods, fields, modifiers, and so on); a discussion on how to obtain these parts follows below. `Java.lang.Class` has several methods for returning information about a class. The return values of many of these methods are instances of classes in the `java.lang.reflect` package. The `java.lang.Class` includes the following methods:

- **Constructor `getConstructor(Class[])`:** returns a constructor with a particular parameter list. The array of Class objects passed into this method corresponds to the types of the parameters in order of the specific constructor. For example, in one class, if we're interested in a constructor with no arguments, we pass in an empty array of Class objects.
- **`int getModifiers()`:** A class declaration may include the following modifiers: `public`, `abstract`, or `final`. The class modifiers precede the class keyword in the class definition. Invoking `getModifiers` returns an integer that can be decoded by the methods of class `Modifier`.
- **`Class[] getInterfaces()`** returns an array of Class objects, each of which corresponds to an interface that the class in question implements. There is no `Interface` class. Instead, interfaces are represented by instances of the class "Class" with the "interface bit" set in their modifiers.

- **Method `getDeclaredMethod(String methodName, Class[] parameterTypes)`**
returns a specific method of a class declared in that class; that is, not inherited from a superclass.

Using these methods of Java reflection, we will discuss how to create the S-Proxy and S-Module classes from original java class.

3.2 Interfaces of S-Module and S-Proxy

As discussed in section 2.2.7, there are some interfaces that the S-Proxy and S-Module must satisfy for hot swapping. There are two interfaces that the S-Proxy and S-Module must implement when they are created from the existing classes.

The `S_ModuleInterface` interface defines the interface that all S-Module classes must implement. An example of the definition of this interface is as follows:

```
Public interface S_ModuleInterface implements SwapInterface,  
                                              IdentityInterface,  
                                              RetriveStateInterface  
                                              MappingStateInterface  
                                              DoesNotUnderstandInterface {  
  
}
```

The `S_ProxyInterface` interface defines the interfaces that all S-Proxy classes must implement. Below is an example of the definition of this interface:


```
Public interface S_ProxyInterface implements NewbehaviorInterface,  
                                           SignalInterface,  
                                           RetriveStateInterface  
                                           SwapControlInterface  
                                           HandleCallExceptionInterface {  
  
}
```

The details of these interfaces implemented by the S-Module and S-Proxy class were discussed in section 2.2.7. The new S-Module and S-Proxy that are created from existing classes should inherit the corresponding existing class and implement the corresponding S_ModuleInterface interface and S_ProxyInterface interface. The following discusses the creation of S-Module and S-Proxy from existing Java class.

3.3 Automatic Creation of Classes of S-Proxy and S-Module

For any S-Component, there will be an S-Proxy and an S-Module. The S-Proxy is an object that represents the S-Module. The S-Proxy provides an interface to the client for the S-Module while hiding the implementation details. First we will create the class Inspector that is used to analyze the original Java class. A discussion of the structure of the SproxyWizard class that is created to create the S-Module and S-Proxy will follow. Finally, a demonstration of the creation of the S-Module and S-Proxy will be included.

3.3.1 Class Inspector

To get across the main points about how to generate the new S-Module and S-Proxy Java code, Reflection is used to analyze an original Java class given as input to ascertain which method signatures the class provides. That information will be used to generate the S-Module and S-Proxy class since the S-Module and S-Proxy supports the same set of methods and has an extension to satisfy the request of hot swapping technique. The class `ClassInspector` is created to hide the details of how Java Reflection is applied to the original class supplied as input.

When creating a new instance of `ClassInspector`, we pass the original Java class as the parameter on to the constructor of `ClassInspector`:

```
ClassInspector inspector= new ClassInspector(Class.forName("CLASS"));
```

Here we use the Java Reflection's `forName` function that loads the original class of a given name, using the current class loader. `ClassInspector` has a set of methods which provides information on the methods and constructors declared on the original class:

```
public String[] getMethods()  
public String[] getMethodCalls(String callInstance, boolean returnFlag)  
public String[] getConstructors()  
public String[] getConstructorCalls()
```

A call to `getMethods()` returns the original class' methods in a form suitable for declaring a new class with the same methods. It starts by making a reflection call:

```
Method[] methods=sourceClass.getMethods();
```

It then iterates through the array of methods and for each obtains the return type, method name, parameter types, and exceptions by using the following invocations:

```
String returnType=thisMethod.getReturnType().getName();  
String methodName=thisMethod.getName();  
Class[] parameters=thisMethod.getParameterTypes();  
Class[] exceptions=thisMethod.getExceptionTypes();
```

All of this information is used to build an output string in the following form:

```
modifiers typeName methodName (parameterList) throws typeNameList
```

Using this string, definitions of methods that reflect the original Java class are created. The `getMethodCalls()`, `getConstructors()`, and `getConstructorCalls()` methods are nearly the same as this. Now we get all information about the original and this information can be used to create the S-Module and S-Proxy.

3.3.2 SproxyWizard

For automatic creation of the S-Module and S-Proxy classes, the basic idea is to produce an `SProxyWizard` utility that will take any given Java class to generate the S-Proxy and

S-Module to construct the S-Component. The SProxyWizard will use the information created by the ClassInspector.

The SProxyWizard class is the entry point in the design. It has a constructor that takes the original Java class:

```
public SProxyWizard(Class originalClass)
```

The SProxyWizard also has a set of methods; each is dedicated to create one aspect of the S-Module and S-Proxy:

```
public void getSProxy()  
public void getSModule()
```

These methods will create class files containing the complete code for the S-Module and S-Proxy based on the original Java class. Within each method, the reflection is used to discover the constructor and method signatures of the original class, and those are then written out in modified form as new Java source code.

3.3.3 Creation of the S-Module and S-Proxy Classes

When creating the S-Module from the original java class, the S-Module must implement the special interfaces that all S-Modules must implement or inherit the existing Java class. The following code shows the main processes in the creation of the S-Module.

```

//create the name of SModule file
sModuleFileName = originalClass().getName() + "SModule.java";
sModuleClassName = originalClass().getName() + "SModule";

//create the S-Module file, write the import files
...
writeImportFiles(myFile);
myFile.write("public class ");
myFile.write(sModuleClassName);
myFile.write(" extends " + originalName + " implements " + S_ModuleInterface + "{\n\n");
myFile.write("\n}");
...
myFile.close();
} catch(IOException e){
    System.out.println("There is error in creating the S-Module File!");
    System.exit(1);
}

```

The name of the S-Module is created from the original class name, with the “SModule” appended. The S-Module will then implement the interfaces that are special for any S-Modules. The source code of the new S-Module will be written to the file that is the original class name appending the “SModule.java”.

The creation of the S-Proxy is more complex than the creation of S-Module since it will forward every method of the original class. When the S-Proxy is created, an instance of ClassInspector is created. First the name of the S-Proxy is created by appending the “Sproxy” to the original class name and the source code file name of S-Proxy is created by appending the “Sproxy.java” to the original class name.

```

//create the name of Proxy file
proxyFileName = originalClass.getName() + "SProxy.java";
proxyClassName = originalClass.getName() + "SProxy";
...

```

The methods that the S-Proxy will forward are then generated by asking the instance of `ClassInspector` for the original class's method calls.

```
returnString = inspector.getMethodCalls(callInstance, true);
```

The code below shows how the source code for the S-Proxy -- including a package statement, S-Proxy import statement, interface declaration, and a set of methods signatures -- is written into an output string based on the information provided by an `inspector` instance. In the S-Proxy, an instance of the S-module is created, named `realInstance`.

```
//create the proxy file
try{
    myFile = new PrintWriter(new FileWriter(proxyFileName, true));
    myFile.write("public class ");
    myFile.write(proxyClassName);
    myFile.write(" extends " + originalName + " implementation S_ProxyInterface{\n\n");
    ...
    myFile.write(" protected " + sModuleClassName + " realInstance = new " + sModuleClassName + " () ";
    ...
    for(int i = 0; i < returnString.length; i++){
        myFile.write(returnString[i]+"\\n");
    }
    ...
    myFile.write("\\n}");

    myFile.close();
} catch(IOException e){
    System.out.println("There is error in creating the PProxy File!");
    System.exit(1);
}
```

Using the Java reflection technique, the automatic creation of the classes of the S-Module and S-Proxy from existing Java class is facilitated. This reduces the time used since

every hot swapping version of the Java class can be created in the same way. There is still a great deal of work remaining to be done. We have to think about the technique details which is currently not supported in this application, such as how to use different creation policies to create different kinds of S-Module and S-Proxy, how to implement the specific methods of S-Module and S-Proxy, etc.

CHAPTER 4. HOT SWAPPABLE SNMP ARCHITECTURE

4.1 Hot Swappable SNMP

In the SNMP system, the management station is typically a stand-alone device. The SNMP agents are hosted in the key platforms such as the hosts, bridges, routers and hubs, so that these resources can be managed from a management station. The SNMP agents respond to requests for information and actions from the management station and may asynchronously provide the management station with important but unsolicited information.

Section 2.3 provided an introduction to SNMP. Since SNMP was first published in 1988, SNMP has become the most widely used network management tool for TCP/IP-based networks. SNMP has been in the dynamic upgrading and extensibility since its publication, from SNMPv1, SNMPv2, SNMPv2u, and SNMPv2* to SNMPv3. The 1993 edition of SNMPv2 included a security facility that was not widely accepted because of a lack of consensus. In 1996, a revised edition of SNMPv2 was issued with functional enhancements but without a security facility. In 1998, SNMPv3 defined a security capability to be used in conjunction with SNMPv2 or SNMPv1. SNMPv3 is modular and is made up of several subsystems. However, SNMPv3 will not be the last version. There are several fields in SNMP which need to be changed or improved to satisfy the new requests.

One field that may be continually changed is the Message Processing Subsystem in SNMPv3. In the SNMP system, the information is exchanged between SNMP entities in the form of a message. Each message includes a message header and a PDU. The Message Processing Subsystem accepts outgoing PDUs from the Dispatcher and prepares these for transmission by wrapping them in the appropriate message header and returning them to the Dispatcher. The Message Processing Subsystem also accepts incoming messages from the Dispatcher, processes each message header, and returns the enclosed PDU to the Dispatcher. If there are any errors in the Message Processing Subsystem, the subsystem must be upgraded without the SNMP entities involved being taken disruptively out of service for recompilation and reinstallation. We may also need to change the Message Processing Subsystem to some new message process subsystems that can handle the new versions of SNMP message processing.

The other important field is that of security enhancement [18]. Management stations execute management applications which monitor and control managed elements. Managed elements are devices such as hosts, routers, bridges, etc., which are monitored and controlled via access to their management information. In this administrative framework, a security model defines the mechanisms used to achieve an administratively defined level of security for protocol interactions. In SNMPv3, there are many security problems which need to be solved. At present SNMPv3 makes use of the User-Based Security Model that includes the DES, MD5, and SHA-1 cryptographic algorithms in order to solve problems related to security. But in the future, as SNMP continues to evolve, the MD5 modules, DES modules and SHA-1 modules might need to be changed

to some other security modules that are as yet unknown. Even the User-Based Security Model can be changed to some currently unknown security model. These changes would have to be coordinated with all corresponding SNMP entities in the SNMP system. If some make use of the new security model and some use the original security model, the whole SNMP system will crash as they cannot communicate.

Applying a new functionality of SNMP to the existing SNMP system involves the recompilation and reloading of hundreds of agents. Usually the evolutions made to the SNMP system have to be done manually, device by device, without the hot swapping technique. This takes a long time and the new features cannot be used until all devices are upgraded. Bearing in mind the above discussion of the trend of SNMP's evolution, we are concerned with dynamically updating the SNMP management. The hot swapping technique provides a flexible method of accomplishing expansive additions. With the hot swapping technique, we can build a global swappable SNMP system so that we can update SNMP entities dynamically in the whole SNMP system without having to recompile the SNMP agents. Using the hot swapping technique, the swapping will become a distributed swapping that is constructed by different local swappings that occur in different SNMP entities in the SNMP system. To ensure the consistency of the application, the distributed swapping takes place in both the SNMP manager stations, and the SNMP agents should be either committed or aborted.

4.2 Hot Swappable SNMP Framework

Figure 4-1. A Swappable SNMP Managed Network

The hot swappable SNMP system is an SNMP system based on the idea of hot swapping techniques. The overall goal is to turn the SNMP system into a dynamic architecture; based on that some parts of the SNMP system can be swapped while the whole system keeps running. The hot swappable SNMP System consists of several key components: managed devices, management stations, swappable SNMP entities (swappable SNMP agents in managed devices, swappable SNMP managers in the management entity), and different swap managers in the management entity and managed devices, which are shown in Figure 4-1.

The swappable SNMP entity is a network-management software module that can reside both in a managed device and in a network manager. A swappable SNMP entity has local knowledge of management information and can translate that information into a form compatible with SNMP. It also has a hot-swapping capability to dynamically change parts of the entity but keep the SNMP entity running. In every swappable SNMP entity there is a swappable SNMP engine. The swappable SNMP engine is an SNMP engine that is partially constructed by some S-Components. There are two kinds of swappable SNMP entities. One is the swappable SNMP manager in the management station and the others are swappable SNMP agents in the managed devices.

The managed device is a network node that contains a swappable SNMP agent and resides on a managed network. Managed devices collect and store management information and make this information available to the NMS using SNMP. Managed devices, sometimes called network elements, can be routers and access services, switches and bridges, hubs, computers hosts, or printers. There is a local swap manager with the swappable SNMP agent.

The management station is a management entity with a swappable SNMP manager. It interacts with SNMP agents by issuing commands (get, set) and by receiving trap messages. The swappable entities in both management entity and managed devices must use the same message processing and security systems to communicate. There is a global swap manager with the swappable SNMP manager.

The swap manager is the core control of the hot swappable SNMP architecture. It acts as a broker and manager between the S-Components and the new S-Modules and controls the distributed hot swapping transaction processes in order to ensure atomic global upgrading. In the hot swappable SNMP architecture, there are two kinds of swap managers that act on different levels. One is the global swap manager that acts in the management station with the swappable SNMP manager; the others are local swap managers that act in managed devices with the swappable SNMP agents. The global swap manager handles the global distributed swapping which is constructed by the local swappings taking place in both the swappable SNMP agents and the swappable SNMP manager. The local swap manager only handles the local swapping process in the swappable agent.

4.3 Swappable SNMP Entities

From the IETF's definition [13], the goal of SNMPV3 architecture is to use encapsulation, cohesion, hierarchical rules, and loose coupling to reduce the complexity of the design and make the evolution of parts of the architecture possible. This makes the SNMPv3 modular and easy to identify, as well as isolates the different modules.

Figure 4-2. A Swappable SNMP Agent

Figure 4-3. A Swappable SNMP Manager

The architecture defined by IETF is composed of an SNMP engine and applications [13, 14, 15, 17 and 19]. This architecture is suited to the use with the hot swapping techniques. In the swappable SNMP system, a swappable SNMP entity is an implementation of this architecture using the hot swappable technique. Each swappable SNMP entity consists of a swappable SNMP engine and one or more associated applications.

Figure 4-2 shows the swappable SNMP agent and Figure 4-3 describes the swappable SNMP manager. Both the SNMP agent and SNMP manager have swappable SNMP engines. A swappable SNMP engine is an SNMP engine with hot swapping capability. It provides services for sending and receiving messages, dispatching messages, authenticating, encrypting and decrypting messages, like the ordinary SNMP engine but with additional abilities which enable some parts of it to be swapped. The swappable SNMP engine in the SNMP Agent has all of the components found in the swappable SNMP engine for the SNMP Manager, plus an Access Control Subsystem. However, this thesis will not undertake a discussion on the dynamic changing of the Access Control Subsystem in the SNMP agent, so the difference between the swappable SNMP engine in the SNMP agent and the swappable SNMP engine in the SNMP manager will be ignored and treated as the same. In our system, we only chose some modules and changed them into the S-Modules. In Figures 4-2 and 4-3, the modules that have been designed as S-Modules are displayed as shaded. There are some other modules that can be candidates for hot swapping, but we did not test them in our swappable SNMP system.

4.4 Swap Managers

In this swappable SNMP system, the global swap manager and the local swap manager behave differently. The global swap manager controls the whole network swapping transaction process and the local transaction process in the swappable manager while the local swap manager controls only the local swap process as it occurs in the swappable agent. Usually in one hot swappable SNMP system there is only one global swap manager, but the number of local swap managers is not specified.

4.4.1 Global Swap Manager

A global swap manager is the core of the swappable SNMP architecture. It controls all of the original S-Modules, local swap managers, local transaction processes and the whole process of distributed swapping. The global swap manager provides the following services:

- **Sending Commands:** The global swap manager sends out commands to let the local swap managers begin hot swapping or stop hot swapping and roll back.
- **Listening Service:** The global swap manager has a listening service that is always listening on a specific port to receive the messages from local swap managers.

- **State Monitor:** The global swap manager has to monitor the states of local swap managers according to the messages received from local swap managers. If some exceptions happen, the global swap manager will send commands to the local swap managers so that they can stop hot swapping and roll back to the original state.
- **Whole Transaction:** The global swap manager has to provide this service. The whole transaction service decides whether the hot swapping process succeeds or fails. Although the whole hot swapping transaction is composed of many local hot swapping transactions, including the local swapping transaction in the management station, the whole transaction has to satisfy the atomicity: that means either the whole transaction succeeds or the whole transaction fails.
- **Time Limit:** The global swap manager should control how long the hot swapping transaction will take. This means that the whole system can only be blocked for the period of time that it takes to finish the hot swapping. The length of time depends on the SNMP system's properties such as scalability, the network delay, etc.
- **Naming Service:** The global swap manager should have access to this service so that all local swap managers can be located by name.

- **Logging Record:** The global swap manager should keep a record of every action and result in the global swapping process.
- **Security Service:** Security is a very important issue in the distributed hot swapping technique. There are several general security concerns regarding the hot swapping technique: ensuring the transmitted data is only received and understood by the local managers, verification of the identity of remote local swap manager, and the authentication of global swap manager.

4.4.2 Local Swap Manager

A local swap manager controls the local swapping process in the swappable SNMP agent; it receives commands from the global swap manager and reports any actions which result. The local swap manager provides the following services:

- **Sending Results:** The local swap manger has the ability to send out results to the global manager to report on the process state so that the global swap manager can make decisions according to this report.

-
- **Listening Service:** The local swap manager has a listening service that is always listening on a specific port to receive hot swapping commands from the global swap manager.
 - **State Monitor:** The local swap manager has to monitor the states of every S-Module so that it can master the local hot swapping process. The local swapping state affects the whole hot swapping process.
 - **Transaction Service:** The local swap manager must provide this service so that the local transactions satisfy the atomicity. This means that the local transaction either succeeds or the whole transaction fails and rolls back to the original state.
 - **Time Limit:** The local swap manager should control how long the local hot swapping transaction will take. This is the time to finish the local hot swapping processing and let zero or close-to-zero down time for the SNMP agent.
 - **Logging Record:** The local swap manager should keep a record of every actions and results in the local hot swapping process.

- **Security Service:** When the local swap manager receive messages from the global swap manager, it has to make sure that the message is really from the global swap manager, the message has not been changed, and the confidential reply can be sent to the global swap manager.

4.5 Swappable SNMP Basic Commands

The communication between the global swap manager and the local swap manager uses four basic commands: PREPARE, SWAP, EXCEPTION, and OK.

- The "PREPARE" command is used by the global swap manager to notify the local swap managers to prepare the swapping issues. When the global swap manager wants to swap some S-Modules in the whole system, it will send the "PREPARE" command with the information of the new S-Module to the local swap managers. The local swap managers will use the information to create an instance of the new S-Module and prepare the swapping process.
- The "SWAP" command is used by the global swap manager to notify the local swap managers to begin the swapping. After the global swap manager sends out the "PREPARE" command and gets the confirmation from the local swap managers that

the preparation has been finished, the global swap manager will send out the "SWAP" command, the local swap managers will begin the swapping and send back the results.

- The "EXCEPTION" message is used between the global swap manager, the local swap manager and the S-Proxy to report any exception events. If any exception happens while the local swap managers are processing the preparation and swapping, the local swap manager will notify the global swap manager by means of an "EXCEPTION" message. The global swap manager will stop the swapping process and send the "EXCEPTION" message to all local managers when it receives any local swap manager's "EXCEPTION" message. The local swap managers will roll back to the original state if they receive an "EXCEPTION" message from the global swap manager.
- The "OK" message is also used between the global swap manager, the local swap managers, and the S-Proxy to report that every event has worked fine. If the local swap manager has finished the preparation and swapping process successfully, the local swap manager will send an "OK" message to the global swap manager. Once the global swap manager has received "OK" messages from all the local swap managers, the global swap manager will send out the "OK" message to the local swap managers. When the local swap managers receive the "OK" message from the global

swap manager, they will unblock all processes, garbage the old S-Module and allow the new S-Module to begin working. Now the whole hot swapping process is finished.

4.6 The Hot Swapping Transaction

A transaction is the execution of a program or a basic unit of work that performs some administrative functions through the use of one or more shared system resources, and results in a very definite, but reversible change in some part of the system properties or states. The most popular and prevalent commit protocol is the Two-Phase Commit (2PC) Protocol [27]. The 2PC protocol is a simple and elegant atomic commit protocol with two types of processes: a single coordinator that decides whether to reach a global commit or abort decision, and the participants that execute the transaction's resource accesses and vote whether to commit or abort. The commit decision is made according to the global commit rule: if even one participant votes to abort the transaction, the coordinator has to reach a global abort decision. If all participants vote to commit the transaction, the coordinator has to reach a global commit decision.

The swapping transaction in our hot swappable SNMP system requires that the swapping must be a unit so that the swapping either succeeds or fails. In the case of dynamic updating of functionality, the global swap manager acts as the coordinator and also

controls the local swapping in the SNMP management station, where the local swap managers are the participants. The global swap manager and local swap managers have to finish the swapping process so that either all of the S-Modules have been swapped successfully, and the entire SNMP system will use the new S-Modules, or the whole swapping process is rolled back and the whole SNMP system will continue to use the original S-Modules.

In a hot swappable SNMP system transaction, a wide range of activities take place. These activities commonly include the following events:

- The global swap manager sends out commands for all local swap managers; controls the global swapping by analyzing the status of the local swap managers; and handles the local swapping in the SNMP manager.
- The local swap manager processes the preparation and initialization of the S-Module; rolls back the swapping; gets and maps the states of the S-Module; “garbage collects” the S-Modules, and controls the S-proxy.

- The S-Proxy will block the services provided by the original S-Module, block the original S-Module, control the access of its S-Modules and get the states of the original S-Module.

All of these actions must co-operate in order to retain the reliability of the whole system. To successfully finish the swapping process, the distributed transaction must be considered. A distributed transaction means bundling multiple operations in which at least two network hosts are involved. In order to satisfy the dependence on an all-or-nothing guarantee for a sequence of operations, the global swap manager must be responsible for creating and managing a global transaction that encompasses all operations against the implied resources. The global swap manager accesses each resource, for example a local swapping process, through the respective local swap manager.

In the hot swapping process, the global swap manager acts as the transaction manager. The local swap managers are joined in the transaction and receive commands from the global swap manager. Now the global swap manager knows about all participants in the transaction and has references for all of them.

The atomicity property of the swapping transaction is the key to maintaining the consistency of a swappable SNMP system. The global swapping in the swappable SNMP system is analogous to a distributed transaction in the sense that these swapping

operations must be grouped together as an atomic unit. The swapping operation must be grouped together into a transaction. Atomicity ensures that either all or none of the swapping operations are performed, so that the SNMP entities in the hot swappable SNMP system use either the new modules or the original modules. If after swapping, some SNMP entities are continuing to use the old modules and some SNMP entities are using the new modules, they cannot communicate and this will destroy the whole SNMP system.

CHAPTER 5. IMPLEMENTATION AND TESTING

5.1 The System Structure

The preceding chapters introduced the architecture and fundamentals of the hot swappable SNMP system. This chapter focuses on implementation and application issues related to the hot swappable SNMP system. The interaction of the global swap manager, the local swap manager and the S-Proxy is illustrated in detail.

As mentioned above, the whole swappable SNMP system can be divided into the management subsystem that is located in the management station, and the agent subsystem that is located in every network device which needs to be controlled. The administrator can manage the whole network through the SNMP management system. The management station is also responsible for controlling the whole swapping transaction. The agent responds to the manager's commands to retrieve values from the MIB, to set values in the MIB, and to handle the local swap transaction. Figure 5-1 shows the decomposition of the swappable SNMP system using UML. The whole swappable SNMP system consists of two different subsystems: the management subsystem and the agent subsystems. The management subsystem consists of the SNMP manager subsystem and the global swap manager subsystem. The agent subsystem consists of the SNMP agent subsystem and the local swap manager subsystem.

Figure 5-1. The Swappable SNMP System

Since the swapping technique is the focus of this discussion, we will analyze the global swap manager subsystem in the management subsystem and the local swap manager subsystem in the agent subsystem. Figure 5-2 shows the classes and their relationship in the global swap manager subsystem that is depicted as a package. The classes in the local swap manager subsystem are depicted in Figure 5-3.

Figure 5-2. The Global Swap Manager Package

Figure 5-3. The Local Swap Manager Package

The global swap manager is comprised of several kinds of classes which deal with the whole swapping transaction, the local swapping transaction, the communication with the local swap managers and the uploading of the S-Module files to the file server so that the local swap manager can download, and so on. The local swap manager deals with the local swapping transaction, obtaining the S-Module codes, communication with the global swap manager, and so on. In the following sections, the swapping sequence diagrams and the state diagrams are used to describe the whole swapping process. The communication between the global swap manager and the local swap manager is discussed in detail.

5.2 The Sequence Diagram of Swapping

By using hot swapping commands, one entire hot swapping process is separated into two parts. The first part of the process involves the global swap manager sending out commands to the local swap managers to prepare for the swapping. In the second part, the global swap manager handles the entire swap and the local swap managers handle the local swapping. Figure 5-4 shows a successful swapping procession.

Figure 5-4. Swapping Sequence Diagram

The preparation process involves the global swap manager sending out information concerning the swapping to the local swap managers in the SNMP agents. The local swap manager will take action to prepare the swapping. The process is as follows:

1. The global swap manager sends "PREPARE" commands to the local swap managers. This command includes the information about the new S-module and the original S-Module that will be swapped. The global swap manager itself will get status from S-Proxy in the management station to prepare the local swapping.

2. When the local swap manager receives this command, the local swap manager begins to prepare the swapping. This is discussed in detail in a later section.
3. The local swap manager will identify the S-Module that is to be swapped, create the new S-Module, get the status of the corresponding S-Proxy, and send back the result to the global swap manager. The different ways that the local swap manager initializes the new S-Module is discussed later. If any errors or exceptions occur during the preparation process, the local swap manager will send an "EXCEPTION" message to the global swap manager; otherwise the local swap manager will send an "OK" message to the global swap manager.
4. The global swap manager will collect the replies from the local swap managers and analyze the reply with the local status from S-Proxy in the management station. If there is an "EXCEPTION" message from any of the local swap managers, or the global swap manager fails the preparation process to get status from S-Proxy in SNMP management station, the global swap manager will send an "EXCEPTION" message to the local swap managers, which will halt the preparation process, and the whole swapping preparation will be interrupted. An example of a kind of exception that one local swap manager has exception is shown in Figure 5-5.

5. If the global swap manager receives an "OK" message from all the local swap managers, the global swap manager will send an "OK" message to the local swap managers, which will finish the preparation process successfully.

Figure 5-5. A Sequence Diagram showing a Failed Preparation

After all the local swap managers have finished the preparation, the local swap managers will be in the prepared successful state. The global swap manager will send out commands to begin the swapping process. The process of hot swapping is as follows:

1. The global swap managers will send out the "SWAP" command to the local swap managers and the corresponding S-Proxy in the management station.
2. When the local swap manager receives the "SWAP" command, the local swap manager begins the swapping. The local swap manager will send "SWAP" commands to the S-Proxy that corresponds to the new S-Module.

3. During the swapping process, the local swap manager will check whether the S-Module is swappable and map the attributes of original S-Module to the new S-Module. If any abnormal event occurs, the local swap manager will roll back the swapping and send an "EXCEPTION" message to the global swap manager. If every event is finished successfully, the local swap manager will send an "OK" message to the global swap manager and wait for the "OK" message from the global swap manager.
4. The global swap manager collects the results from the local swap managers.
 - If the global swap manager receives "OK" messages from all local managers and the swapping is successful in the management station, it decides to commit to the swap transaction and sends an "OK" message to all local swap managers. The local swap manager will send an "OK" command to the corresponding S-Proxies, unblock all the processes, garbage collect the original S-Modules, and the swapping process will be complete.
 - If the global swap manager receives an "EXCEPTION" message from any of the local managers or the local swapping fails in the SNMP management station, it decides to abort the swap transaction and sends an "EXCEPTION" message to all local swap managers. The local swap managers will send an "EXCEPTION" command to the S-Proxies to abort the swapping and roll back to the original state. The entire swapping process will then have failed. An example of this kind of exception that happens in local swap manager is shown in Figure 5-6.

- If the global swap manager does not receive any messages from one or more of the local managers after the timeout period has expired, it decides to abort. The global swap manager will send an "EXCEPTION" message to all local swap managers, which will then abort the swapping process and roll back to the original state. The whole swapping process will then have failed.

Figure 5-6. Sequence Diagram showing a Failed Swapping

5.3 Hot Swapping State Diagrams

In the hot swappable SNMP system, the global swap manager, the local swap manager and the S-Proxy communicate with each other to control the whole swapping process. This section analyzes their finite state machines when they are running in the swapping process.

Figure 5-7. State Diagram of Global Swap Manager

Figure 5-7 shows a global swap manager state diagram. The initial state of this FSM is Idle. When the global swap manager begins the swapping process, it sends out the PREPARE command with information about the new S-Module, the original S-module and the corresponding S-Proxy to the local swap managers and get the status from S-Proxy in the management station. The FSM then enters the WaitingPreparing state. This FSM waits to collect the information from the local swap managers in the SNMP agents and the S-Proxy in the SNMP manager regarding the swapping preparation; the information received will either be "OK" or "EXCEPTION". If, within a certain period of time, all the collected information is "OK" and the local preparation in the SNMP manager is successful, then the FSM enters the PreparedSuccessful state. If even one "EXCEPTION" is received, or some replies are not received within the time constraint, or local preparing in the management station fails, the FSM will send out an "EXCEPTION" to all local swap managers and return to the Idle state. In the PreparedSuccessful state, the global swap manager will send the "SWAP" command to all local swap managers and enter into the WaitingSwapping state. Now the FSM will collect the swapping information from the local swap managers in the SNMP agents and the S-Proxy in the SNMP manager. If all replies are "OK" and the local swapping in the SNMP manager is successful, the global swap manager will send an "OK" message to all local swap managers, finish the whole swapping process and return to the Idle state. The whole swapping process has then been completed successfully. If any reply is an "EXCEPTION" message, or any replies do not arrive within a certain period of time, or the local swapping fails, the global swap manager will send out an "EXCEPTION" and return to the Idle state. The whole swapping process has then failed.

Figure 5-8. State Diagram of Local Swap Manager

There are five states in the local swap manager, as shown in Figure 5-8. In the Idle state, the local swap manager listens to a special port to get the commands from the global swap manager. When the “PREPARE” command is received, the local swap manager

enters the Preparing state. In this state, the local swap manager will dispatch the “PREPARE” command, use the information received to initialize the new S-Module, identify the original S-Module and S-Proxy, and get the status of the S-Proxy. If these actions are successful, the local swap manager will enter the FinishedPreparing state. In the FinishedPreparing state, if the local swap manager receives an “EXCEPTION” from the global state manager, the local swap manager will return to the Idle state. In the FinishedPreparing state, if the local swap manager receives a “SWAP” command, it will send the “SWAP” command to the S-Proxy and enter the Swapping state. In this state, if the local swap manager gets an “OK” from the S-Proxy, it knows that the S-Module is in the swappable state, and the S-Proxy will return the states of the original S-Module. The local swap manager will map the states to the new S-Module, and swap the S-Modules. If this swapping process is successful, the local swap manager will send an “OK” reply to the global swap manager and enter the Finished state. If the local swap manager gets an “EXCEPTION” from the S-Proxy or it fails in the process of mapping states and swapping, it will send an “EXCEPTION” reply to the global swap manager, roll back the swapping and garbage collect the new S-Module. In the Finished state, if the local swap manager get an “OK” command from the global swap manager, it will send an “OK” to the S-Proxy and return to the Idle state. The whole swapping process has then completed successfully. If the local swap manager gets an “EXCEPTION” from the global swap manager, it will send an “EXCEPTION” to the S-Proxy, roll back the local swapping and return to the Idle state.

Figure 5-9. State Diagram of S-Proxy

The S-Proxy gets its swapping instructions from the swap manager; Figure 5-9 shows its swapping states. In the SNMP manager, the S-Proxy communicates with the global swap manager; in the SNMP entity, it communicates with the local swap manager. In the Idle state, the S-Proxy does not handle any request from its clients, whereas in the Busy state, the S-Proxy does handle requests from its clients. In the Idle and Busy states, there is always a thread that listens on the common port for messages from the swap manager. When the “SWAP” message is received, the S-Proxy will block its services and enter the

Preparing state. In the Preparing state, the S-Proxy will check whether the original S-Module is in a swappable state and try to get the attributes of original S-Module. If the S-Module is not swappable or failed to get the attributes, the S-Proxy will send an "EXCEPTION" to the swap manager and return to Idle or Busy state. If the S-Module is swappable and all the attributes of the original S-Module are obtained successfully, the S-Proxy will return the attributes to the swap manager and enter the Swapping state. In the Swapping state, the S-Proxy will wait for the command from the swap manager. If the S-Proxy gets an "OK" command from the swap manager, it will change the reference to the new S-Module and return to the Idle or Busy states, depending on whether any services are blocked. If the S-Proxy gets an "EXCEPTION" command from the swap manager, it will block the original S-Module and return to the Idle and Busy states.

5.4 Issues Concerning the New S-Module Transfer

When the global swap manager sends a command to the local swap managers to prepare the new S-Module, this command includes information about the new S-Module. The local swap managers will use this information to find and instantiate the new S-Module. In our experiments, the code of new S-Modules has been installed in the SNMP agents so that the local swap managers only need to know the name of S-Module to instantiate it. However, in the hot swapping system, the new S-Module may need to be sent out from the managing station to all the participants. In the following, we discuss two

approaches to deliver the new S-Modules. The first one makes use of java serialization. The other uses file deployment.

5.4.1 Java Serialization

The Java language is strongly network-oriented and provides some support for the mobility of code [21]. Since the JDK version 1.1, Java Serialization has been considered to be the default persistence mechanism for the Java language. Java Serialization is part of the standard `java.io` package and provides facilities to write/read (serialize/ deserialize) objects to and from byte streams. A byte stream is an abstract representation, to which a disk file or a network connection may physically correspond. There is no way to change objects on the byte stream individually, and the entire byte stream should be read into memory and then written back to make any changes. For reasons of security, an instance can be serialized only if its class implements the standard `java.io.Serializable` interface. Many of the Java core classes do not do this. There are no methods in the `java.io.Serializable` interface; the interface just serves as a marker to say that a class can be serialized. Using Java Serialization, the S-Module information can be written into the byte stream and be sent to the local swap manager. Once the local swap manager gets this information, the local swap manager can instance the new S-Module.

However, there are some shortcomings with Java serialization. For example, just adding `Serializable` implements to a class definition doesn't automatically make a class serializable. All the instance variables of the class must be serializable too. If they aren't

serialized, an exception is thrown. To mark an instance variable as not for serialization, the keyword "transient" is added to the definition. Classes containing platform-specific implementation information are not serializable and should be marked as transient. If objects to be stored and retrieved refer to other objects, those other objects must be stored and retrieved at the same time to maintain the relationships between the objects. When an object is stored, all of the objects that are reachable from that object are stored as well. This request that the S-Module must satisfy is the "serializable" request.

5.4.2 File Deployment

With regard to the file deployment method, there are two applicable cases. One case involves the local swap manager first downloading the S-Module file, then compiling the file, and finally using its own ClassLoader to load the new S-Module to the system. In the other case, the local swap manager uses URLClassLoader to download the S-Module class that can be specified completely with a URL (for example a file:, http:, or jar: URL).

In the first case, the new S-Module file is stored in a file server. The file server will handle the request to download these files. In the preparation portion of the hot swapping, the global swap manager sends the location information for the new S-Module to the local swap managers using the "PREPARE" command. When the local swap manager gets the location information, it contacts the file server and downloads the S-Module files from the file server.

After downloading the S-Module files from the file server, the local swap manager will first compile the file, and then the ClassLoader will load them into the system. The responsibility of ClassLoader is to do the loading and verification. The ClassLoader finds and loads the byte codes for the class definitions. Once loaded, they are verified before the ClassLoader can create actual classes. In addition to finding, loading, and verifying, the ClassLoader performs many other security-related duties. First, the ClassLoader will not attempt to load any classes in java.* packages from the network. This ensures that the JVM isn't tricked into using false representations of the core class libraries--ones that could break the Java security model. Second, the ClassLoader provides separate name spaces for classes loaded from different locations. In this way, classes with the same name loaded from different hosts will not clash. In addition, classes loaded from different hosts will not communicate within the JVM-space since this may permit untrusted programs to get information from trusted ones.

In the second case, the URLClassLoader is used to create custom class loaders. URLClassLoader is a special subclass of the newly introduced SecureClassLoader, where SecureClassLoader allows associated permissions based on the source of a loaded class. With the URLClassLoader, any class or location that can be specified completely with a URL (for example a file:, http:, or jar: URL) can be loaded directly through a URLClassLoader. To provide an operation like encryption or load class bytes directly out of a database, the URLClassLoader can be subclassed and extended for specific requests or specific protocol handlers. Through the use of this method, the global swap manager

only needs to put the S-Module classes in the file server, then tell the local swap manager where to find these classes. The local swap manager can load the S-Module from the file server using the `URLClassLoader`.

5.5 The Client/Request Structure

During the preparation and swapping periods, there are many components of hot swappable SNMP software that are used to manage the distributed swapping. It is vital to keep the connection between the global swap manager and the local swap managers. In the swapping and preparing phases, the global swap manager must keep track of all the current local swap manager's connections, check their connection status, process their replies and recoup the replies once a connection is terminated. The global swap manager will send commands to the local swap managers and get the replies from the local swap managers once they have finished the requested events. From the point of view of client-server application, in every global swap manager and local swap manager pair, the global swap manager is on the client side and the local swap manager is on the server side. The global swap manager will send the command to the local swap manager and wait for the result sent back from local swap manager just like the client sends request to the server and waits for the server's result. Figure 5-10 shows the corresponding server/client structure.

Figure 5-10. The Client/Server Structure

To ensure that the hot swapping process is working successfully, the local swap managers must receive the commands from the global swap manager and the global swap manager must receive the replies from the local swap managers. So the communication between clients and server becomes very important. In this case, TCP is chosen because it is a connection-oriented protocol.

TCP handles packet flow between systems. Using TCP, an end-to-end connection must be established before data exchange can occur. Data bytes are delivered in sequence, the delivery of data is guaranteed, and the connection is terminated once the data has been transferred.

An intuitive first step towards the first objective is to separate out the client/server connections and requests from the swap managers and give them their own classes. Since the global swap manager will connect with multiple local swap managers at the same time, the connection class has to be multithreaded. In this case, the class `Global_SendMessage` is used to implement the interface `Runnable` that the global swap manager can use to connect the local swap managers. The local swap manager needs to receive commands from the global swap manager so that it has to monitor a special port all the time. In this case, a `Local_GetMessage` class is created to implement the interface `Runnable` that the local swap manager can use to reply to the global swap manager. Both the `Global_SendMessage` and `Local_GetMessage` classes construct a thread for themselves by calling its own `start()` method and execute its central process from that point, thereby keep the swap managers free of the connection details.

Figure 5-11. The Class Diagram for the `Global_SendMessage`

Figure 5-11 shows the most important part of the `Global_SendMessage` class diagram.

The `Global_SendMessage` has the following instance variables:

- `processSocket`: This is a `Socket` that includes the computer name and port number through which the global manager can send commands to the local managers. In this way, a connection-oriented link can be set up between the global swap manager and the local swap manager.
- `replyVector`: This contains the identifies of the computers in which the local managers run. The local swap manager will register at the global swap manager.
- `command`: This contains the commands sent to the local manager by the global manager.
- `dispatch()`: This function is used to process the commands to be sent and received. The outgoing messages will be dispatched so that the local swap manager can have information concerning how to transfer the new S-Module, which S-Proxy will be chosen, and so on. The incoming messages will be dispatched so that the global swap manager can know the local swap manager's response.

- `Decryption()` and `encryption()`: This can be used to improve the security of the communication between the global swap manager and the local swap manager.

Global manager uses the `sendCommand()` method to send commands to local managers. This method first gets the computer's identify from the vector that contains the local managers. Next, it creates a socket object with the computer name and the port number (4444), through which the local manager can receive the global manager's command. It then creates an instance of a `Global_SendMessage`, using the socket object and command. Now the connection between the global swap manager and local swap manager is built for every local swap manager, and they are handled at the same time.

Figure 5-12. The Class Diagrams for the `Local_GetMessage`

Figure 5-12 shows the class diagram of `Local_GetMessage`. The `Local_GetMessage` has the following instance variables and methods:

- `processSocket`: This is a `ServerSocket` that knows the port number when the local manager is listening to commands sent by the global managers. It will create a connection-oriented link with global swap manager.
- `respondVector`: This is used to store the responds that the local swap manager sends back to the global swap manager concerning the swapping results.
- `listening()`: The `Local_GetMessage` is always listening to a special port which is waiting for the commands sent from the global swap manager.

The local manager uses the `listening()` method to monitor the global manager's command. This method first creates a `ServerSocket` object with the port number (4444) where the global manager will send a command. Making use of the socket object and other objects then creates an instance of `Local_GetMessage`. The object of the `Local_GetMessage` will run to dispatch the commands received from the global swap manager and the results sent to the global swap manager. The local swap manager will process the swapping according to this dispatched information.

5.6 Testing

The architecture of the hot swappable SNMP system is implemented. When the SNMP manager broadcasts a message to the agent, the SNMP agents interpret the SNMP message and receive request from the SNMP manager. The management information that is used in the swappable SNMP management system is collected and stored by the agents and made available to the manager. Polling and event reporting are used to make the agent information available to the manager. Polling is a request-response interaction between a manager and agent. The manager can use polling to query any agent and request the values of various information elements. The hot swapping protocol uses TCP to communicate between the global manager and local manager. Using TCP solves the reliability problem since it makes sure that local managers can receive the commands from the global manager and the global manager can get the local manager's result. The hot swapping command is used among the global swap manager, local swap manager and S-Proxy to communicate the swapping status. The whole hot swapping process must have atomicity, either committing to the hot swapping or aborting the hot swapping.

The swappable SNMP application was designed to have one SNMP manager and several SNMP agents. The global swap manager is registered with the SNMP manager and includes the new S-Modules. The global swap manager is designed to run on another thread to control the global hot swapping services and provide the local swapping service in the management station for this application. The local swap managers are registered in

those corresponding SNMP agents and run on another thread to provide the local hot swapping services. Once the agents start up, the local swap manager will start running in a separate thread. When the agents get the information from the manager, the local swap manager will register to the global swap manager.

In this test, the swappable parts of the SNMP manager and SNMP agents are constructed as S-Components. Each S-Component is composed of an S-Proxy and an S-Module as defined in hot swapping technique. Only the S-Modules are swappable. These S-Components are discussed in section 4.3. In our hot swappable SNMP system, there are the following S-Components: Message Processing S-Component (s-proxyv3mp and v3mp), User Security Model S-Component (s-proxyUSM, user security model), MD5 S-Component (s-proxy MD5, HMAC-MD5-96), and DES S-Component (s-proxyDES, DES).

```

4  // Create a request
5  // Create a request
6  // Create a request
10 // Create a request
11 // Create a request
12 // Create a request
13 // Create a request
14 // Create a request
15 // Create a request
19 // Create a request
20 // Create a request
21 // Create a request
22 // Create a request
23 // Create a request
24 // Create a request
25 // Create a request
26 // Create a request
27 // Create a request
28 // Create a request
29 // Create a request

```

Figure 5-13. The SNMP Manager Sending Request and Processing Response

Figure 5-13 displays the procedure where the original SNMP manager sends a request to the SNMP agent and receives a response from the SNMP agent before swapping begins. Line 4 shows the SNMP manager beginning the process and first sending the request to the Dispatcher. Once the Dispatcher has processed it, the request is sent to the Message Processing Model S-Component. The proxy forwards the service to the real Message Processing Model, v3MP (line 5, 6). Once the Message Process Model S-Component has finished the process, the request will be sent to the User Security Module S-Component (line 10, 11). First the DES S-Component will encrypt the request (line 12, 13). Then the MD5 S-Component will deal with the authentication (line 14, 15). When all of these actions have been finished successfully (line 19), the SNMP manager sends out the request (line 20).

```

3  // Get the request and the request must first be extracted. The Message Processing Model S-
4  // Component completes this extraction. The SNMP agent uses the same Message
5  // Processing Model S-Component as the SNMP manager (line 4, 5). The User Security
7  // Module S-Component will then check the security of the data (line 7, 8). The MD5 S-
8  // Component is used to make the authentication check (line 9, 10) and the DES S-
9  // Component is used to make the authentication check (line 9, 10) and the DES S-
10 // Component is used to make the authentication check (line 9, 10) and the DES S-
11 // Component is used to make the authentication check (line 9, 10) and the DES S-
12 // Component is used to make the authentication check (line 9, 10) and the DES S-
13 // Component is used to make the authentication check (line 9, 10) and the DES S-
16 // Component is used to make the authentication check (line 9, 10) and the DES S-
17 // Component is used to make the authentication check (line 9, 10) and the DES S-
18 // Component is used to make the authentication check (line 9, 10) and the DES S-
20 // Component is used to make the authentication check (line 9, 10) and the DES S-
21 // Component is used to make the authentication check (line 9, 10) and the DES S-
22 // Component is used to make the authentication check (line 9, 10) and the DES S-
23 // Component is used to make the authentication check (line 9, 10) and the DES S-
24 // Component is used to make the authentication check (line 9, 10) and the DES S-
25 // Component is used to make the authentication check (line 9, 10) and the DES S-
26 // Component is used to make the authentication check (line 9, 10) and the DES S-
27 // Component is used to make the authentication check (line 9, 10) and the DES S-

```

Figure 5-14. The SNMP Agent Sending Response

A discussion on the procedure whereby the SNMP agent receives the request, processes the security, gets the request, and creates a response follows below. Figure 5-14 displays this procedure as carried out by the SNMP agent. Line 3 shows that the SNMP agent gets the request and the request must first be extracted. The Message Processing Model S-Component completes this extraction. The SNMP agent uses the same Message Processing Model S-Component as the SNMP manager (line 4, 5). The User Security Module S-Component will then check the security of the data (line 7, 8). The MD5 S-Component is used to make the authentication check (line 9, 10) and the DES S-Component is used to make the authentication check (line 9, 10) and the DES S-Component is used to decrypt the request (line 11, 12). The User Security Module, DES model and MD5 model are the same as those used in the SNMP manager. If the security check is successful, the SNMP agent gets the request and will create the response. First

the response is created (line 16); then the response is sent to the Message Processing Model S-Component to process (line 17, 18); the User Security Module S-Component will handle the security problem (line 20, 21); the DES S-Component is used to decrypt the message (line 22, 23); and the MD5 S-Component is used to achieve the authentication (line 24, 25). If this process is successful (line 26), the SNMP agent will send out the response to the SNMP manager (line 27).

The procedure whereby the SNMP manager handles the SNMP agent's response is shown in Figure 5-13. When the SNMP manager gets the response (line 21), the Message Process S-Components will process the response (line 22, 23). The User Security Module S-Components will then check the security of the response (line 25, 26). The MD5 S-Component is used to take the authentication check (line 27, 28) and the DES S-Component is used to decrypt the request (line 29, 30). If all the security checks are successful (line 31), the SNMP manager gets the response from SNMP agent (line 32).

```

MS-DOS - java swappableSnmpEngine.Global_MasterApplication
4  ...
5  ...
6  ...
10 ...
11 ...
12 ...
13 ...
14 ...
15 ...
19 ...
20 ...
21 ...
22 ...
23 ...
24 ...
25 ...
26 ...
27 ...
28 ...
29 ...

```

Figure 5-15. The SNMP Manager Sending a Request after Swapping

```

Command Prompt - java swappableSnmpEngine.Local_MasterApplication
3  ...
4  ...
5  ...
7  ...
8  ...
9  ...
10 ...
11 ...
12 ...
13 ...
16 ...
17 ...
18 ...
20 ...
21 ...
22 ...
23 ...
24 ...
25 ...
26 ...
27 ...

```

Figure 5-16. The SNMP Agent Sending a Response after Swapping

Figure 5-15 and Figure 5-16 show the test result after some S-Modules have been swapped, and the procedures whereby the SNMP manager sends out the request, the

SNMP agent processes the request and creates the responses, and the SNMP manager receives the response, processes the response and gets the result. These procedures are the same as discussed above. The only difference is that the User Security Model S-Component has been changed to “s-proxyUSM” and “My USM1”; the MD5 S-Component has been changed to “s-proxy MD5” and “My MD6”; and the DES S-Component has been changed to “s-proxyDES” and “My DES1”. Please note that the proxies in those S-Components remain the same, but the S-Modules are swapped. This test shows that after the S-Modules have been swapped, the whole hot swappable SNMP system works well. The SNMP manager will use the new User Security Module, “My USM1”, to provide the new security function; the new module “My MD6” is used to achieve the new authentication and the new module “My DES1” is used to provide the new decryption. The SNMP agents use the same changed modules to handle the request of the SNMP manager and process the response. The entire swappable SNMP system was not aware of the swapping transaction.

Figure 5-17 shows the swapping of the MD6 module with the MD5 module in the SNMP management station. The management station keeps polling the agents when the global swap manager swaps the modules. It is evident from the diagram that the system uses the MD5 module before the swapping, but the system changes to use the MD6 module after the swapping. A similar swapping process will occur in the SNMP agent.

Figure 5-17. Swapping in The Polling Process

The test results show that when the local swap manager begins the swapping transaction, the affected parts are blocked, but the other parts of the agent are still running and processing the message. The swapping did not interrupt the communication between the SNMP agents and the SNMP manager. The SNMP manager keeps sending out SNMP requests and does not notice the swapping.

The following will demonstrate the swapping procedure for the S-Modules in a hot-swappable SNMP system. In Figure 5-18, the status of the global swap manager is displayed. In this figure, the new S-Modules are listed on the left side, and the corresponding original S-Modules are listed on the right side. When a new S-Module will be swapped, it is selected and displayed in the middle area. The swapping log is displayed. From the log, we can see the results of the swapping: how long each local swap manager takes to prepare the swapping, how long each local swap manager takes to finish the swapping, and when the whole swapping process is finished. In Figure 5-18, the new S-Module MyMD9 is used to swap the original S-Module SNMPv3MP. When the “Prepare” button is clicked, the global swap manager will send out a “PREPARE” message to the local swap managers. After the local swap managers finish the preparation, the result is sent back. In our test, the cost time of preparing and swapping are displayed as well. Figure 5-18 shows the times that the local swap managers used to prepare the swapping; the longest is from 134.117.60.65 and takes 30 milliseconds. After the global swap manager has checked that the preparation result is correct, the global swap manager will begin the swapping procedure. Now the global swap manager will send a “SWAP” command to the local swap managers.

Once the local swap manager has finished the swapping, the result will be sent back to the global swap manager. In Figure 5-18, we saw that all the local swap managers finished the swapping successfully. The local swap manager 134.117.60.65 took the longest time - 30 milliseconds - to finish the swapping. Now the swapping is finished and

the whole system will use the new modules. The entire swapping process was completed in 180 milliseconds.

Figure 5-18. The Global Swap Manager

In Figure 5-19, the status of the local swap managers are displayed. When the local swap manager has received the commands from the global swap manager, after preparing the swapping process, the selected new S-Module is indicated. The result of the preparation, the transaction, and the swapping of the new S-module are displayed. In Figure 5-19, the

local swap manager is on host 134.117.60.63. When the local swap manager has finished the preparation successfully, it will send the result to the global swap manager. When the local swap manager receives the “SWAP” command, it will begin the swapping. Figure 5-19 shows that the local swap manager has successfully finished the swapping of the new S-Module, MyDES2. For a new swapping procedure, the local swap manager has finished preparing the new S-Module MySHS3 successfully.

Figure 5-19. The Local Swap Manager

In testing the swappable SNMP system, we want the SNMP system to provide continuous service while allowing the SNMP engines to adapt to new requests and configuration changes. The SNMP management station has to poll the agents to receive the collected information from the managed devices. The polling interval is user-defined and depends on the number of devices being managed; it can be as short as 3 minutes to as long as one hour or more. The polling interval for SNMP is generally longer since the act of polling uses network bandwidth to poll and actually transfer information. The traffic on the network is important for global swapping since the global swap manager controls the whole swapping and depends on the information collected from the local swap managers.

When we dynamically update the SNMP engine by swapping S-Modules in the Message Processing Subsystem or the Security Subsystem, we need to block the processing of messages, swap the S-Modules and then resume the blocked processing of messages. Once the processing has been resumed, there exist two kinds of messages: the old messages are created by the original module and the new messages are created by the new module. The problem is that the new version of the SNMP engine may not understand the old messages created by the old version. This causes an incompatibility problem. To solve this problem, the swapping of S-Modules should not influence the processing of these old messages. It means that the old messages should be processed by the original module and the new messages should be processed by the new module so that no incompatibility problem occurs. First we will discuss the advantages of swappable SNMP architecture, the differences between SNMP versions and how to distinguish the messages created by different SNMP versions. We will then discuss how to solve this

incompatibility problem by adding extra modules to the SNMP agent as simple placeholders in the compilation time, and swap these extra modules in the run time. This approach can be implemented in the future work.

As discussed in section 4.3, the swappable SNMP engine contains a Message Dispatcher, a Message Processing subsystem, and a Security subsystem. The advantage of this architecture is that the Message Dispatcher allows for concurrent support of multiple message processing modules and the message processing module allows for concurrent support of multiple security modules. The Message Dispatcher sends and receives SNMP messages to/from the network, determines the version of the SNMP message and interacts with the corresponding message's processing Module. The message processing module then checks the value of the msgSecurityModel and sends the message to the corresponding security modules. In this way, the Message Dispatcher keeps track of all message processing modules and the message processing module keeps track of all security modules. In this architecture, the different versions of the SNMP agent all have the same Message Dispatcher, but implement different message processing modules or different security modules. When we want to update the functionality of an SNMP agent from one version to the other version, we only need to update the relevant message processing module or security module, and the Message Dispatcher remains unchanged.

The information exchanged between SNMP entities contains fields for global data (such as the SNMP version, the message identifier, the maximum message size, the security model and the level of security), fields for the parameter of the security model, fields for the naming scope (context identifier and name), and the PDU. The message structure is illustrated informally in Figure 5-20; the shaded fields are processed by the Message Processing Subsystem.

Figure 5-20. SNMPv3 Message Format

The SNMP version identifier, `snmpVersion`, identifies the version of the message processing module in use. Different message processing modules are identified by different `snmpVersion` values. The security model identifier, `msgSecurityModel`, identifies the security module in use. The `msgSecurityModel` is an identifier within the range of 0 through $2^{31} - 1$, whereas a different security module has a different `msgSecurityModel` value.

From the above analysis, we know that the swappable SNMP agent supports multiple message processing modules and security modules; the information created by different

SNMP versions can be distinguished from its `snmpVersion` or `msgSecurityModel`. In this case, we add an extra message processing module and an extra security module in the compilation time as placeholders in the SNMP agent. These extra modules are inactive, but the Message Dispatcher knows the `snmpVersion` of the extra message processing module and the message processing modules know the `msgSecurityModel` of the extra security module. During run time, when we want to update the SNMP engine dynamically by swapping the message processing module or the security module, we swap the corresponding extra module instead of the module that we actually want to swap. Since the swapping only involves the new S-Module and the corresponding extra module, the processing of old messages is not affected. This solves the incompatibility problem. To add the extra modules in the SNMP agent at compilation time, we do the following:

- Create the extra message processing module and the security module. In our swappable SNMP agent, those modules that have some common methods and attributes are grouped under the same parent class. The extra message processing module has the same parent class as the original message processing module, and the extra security module has the same parent class as the original security module.
- Identify the extra message processing module and the extra security module in the SNMP agent. With regard to the Message Processing Subsystem, we label the `snmpVersion` of this extra message processing module as 4 and attach it to the

Message Dispatcher so that the Message Dispatcher can identify the extra message processing module by checking the `snmpVersion`. The same process is applied to the Security Subsystem. We put the extra security module in the Security Subsystem and assign the `msgSecurityModel` of this extra security model to be $2^{31} - 1$ so that the message processing modules can identify the security module by checking the `msgSecurityModel`.

- Deploy these kinds of swappable SNMP agents in the network. In each swappable SNMP agent, there are three message processing modules - the SNMPv1's message processing module, the SNMPv3's message processing module and the extra message processing module; and there are four security modules- the MD5 authentication module, the SHA authentication module, the DES encryption module and the extra security module. However, the extra message processing module and the extra security module are inactive.

The swapping of the new S-Module and extra module follows the same procedure described in section 5.2. Since no active modules are blocked during the swapping, the processing of messages will not be influenced during the swapping. The global swap manager handles the swapping transaction of the new module with the extra module. If the global swapping is successful, the new module will become active and be used to create the messages; the identifier of `snmpVersion` or `msgSecurityModel` in the new message will show that the update has occurred successfully. When the SNMP management station wants to send a request, if the Message Dispatcher finds the value of

snmpVersion is 4, the Dispatcher will forward the message to the new message processing module. If the message processing module finds the value of msgSecurityModel is $2^{31} - 1$, it will forward the message to the new security module. When the SNMP agent receives the request, its Message Dispatcher will check the SNMPVersion and msgSecurityModel and forward the request to corresponding modules. Now the new message created by the new SNMP engine can be processed by the new S-Module.

This approach adds extra S-Modules and swaps the new S-Module with the corresponding extra module in the SNMP agent. The advantage of this approach is that the original S-Module can exist concurrently with the new S-Module and be protected from being affected by the swapping. Once the blocked processing has been resumed, the original modules will handle the old messages and the new module will handle the new messages. In fact, after the swapping, the original module becomes an extra module, since the new module will handle the processing of messages after all old messages have been processed by original module. If the need arises to update SNMP engine again, we can swap the new S-Module with the new extra module.

5.7 Scalability of Swappable SNMP System

In order to monitor the changing rate of global swapping time when the number of swappable SNMP agents increases, we built an LQN (Layered Queuing Network) [35] model to simulate the swapping processes. The LQN is briefly introduced in section

5.7.2. First, we tested the swappable SNMP system in a local network consisting of 13 computers. Then, we use the LQN model to simulate the scalability of swappable SNMP system; the parameters in the LQN model are set according to the tested result.

5.7.1 Tested Result of the Swappable SNMP System

Our experiments focused on the length of the global swapping time. We tested the swapping process and measured the average time of global swapping of S-Modules in the swappable SNMP system. The test was run on a local network consisting of 13 computers – each with a Pentium III/500MHz CPU and 128mb ram.

In this test, we tested the swapping of the following S-Modules: the MD5 S-Modules, DES S-Modules, SHA-1 S-Modules, and the Message Processing S-Modules. The global swapping time does not include the time that the swap managers used to prepare the swapping since the code of new S-Modules has been installed in the SNMP agents in our test. In our test, the global swapping time is measured from the global swapping manager sends out “SWAP” command until the global swap manager finishes the global swapping. The local swapping time in the SNMP agents is measured from the local swap manager receives the “SWAP” command until the local swap manager sends out the result to the global swap manager. We tested the swapping time in two cases: case one is that the SNMP management station handles polling and the global swap manager handles the swapping at the same time; case two is that only the global swap manager handles

swapping. In case one, the polling frequency is 2 seconds. The tested result is shown in Table 5-1.

# of swappable SNMP agents	2	3	4	5	6	7	8	9	10	11	12	13
Swapping with Polling time (ms)	130	195	240	270	300	340	360	370	400	420	435	452
Swapping with non- polling time (ms)	74	95	130	155	180	195	211	220	237	250	267	275

Table 5-1. Average Time of Global Swapping

We can see that the global swapping time in case one is longer than case two. This is because that in case one, the SNMP management station used polling to collect the regular information about the swappable SNMP agents, the swappable SNMP agents had to process the responses. When the local swap manager, which is located in the swappable SNMP agents, received the “SWAP” command from the global swap manager, it could not start the swapping until the original S-Module had finished its action in the processing of responds so that the new S-Module could swap the original S-Module. In our test, the average of local swapping time in the SNMP agents is around 70 ms in case one and 30 ms in case two. The results are plotted in Figure 5-21 along with the simulated results. Since our system is constructed with only 13 computers, the network delay is less than 1 ms in ideal situations. So the network delay is very small compared to the local swapping time, and therefore local swapping time plays a major role in the length of global swapping time. With the increase of swappable SNMP agents, if the SNMP management station collects regular information by polling, the congestion of the network will increase. The performance of the whole swappable SNMP

system will mainly depend on the processing speed of the management station and the congestion in the network.

5.7.2 LQN Model

According to the architecture of the swappable SNMP system, we build the LQN simulation model of the swappable SNMP system to estimate the global swapping time for a large swappable SNMP system and set the parameters of this LQN model according to the test results.

The LQN model can be used to describe a system by the sets of resources that are used by its operations. Every operation requires one or more resources, and the model defines a resource context and an architecture context for each operation. The architecture context is a software object which executes the operation. The resource context is a set of software and hardware entities required by the operation. To define a system, the LQN uses tasks which are objects with resource properties. A task can model an object which provides operations, a resource, or both of them. A task offers one or more services through methods called entries, and these entries execute and make calls to other entries. LQN can be used to model the distributed network system.

Only the successful swapping scenario is described in this LQN model. In this LQN model, we assume that all local swap managers finish the local swapping within the same period of time. The task “user” in our LQN model is used to simulate the global

swapping manager which initializes the swap request. We use task “managerT” to represent the demands for the manager which creates the sending thread and release command. Therefore, there are two entries in this task. One is the entry “presend” that the global swap manager uses to create the sending thread that is used to connect the local swap manager. The other is the entry “release” that the global swap manager uses to send a swap decision to the local swap managers. Since the global swap manager uses multithread to prepare and send commands to the local swap managers and get the results from the local swap managers, in our LQN model, we use LQN’s activities to model the parallel execution. Activities simulate the situation that the manager has to wait for all the responses from the local swap managers before sending out a decision to undertake the distributed swapping. Therefore, a join operation is required. We use the entry “proc” in the task “procT” to describe the global swap manager’s processing of the replies from the local swap managers. The network delay of the replies is included in the entry “proc”. Tasks “SwapnetT” and “relnetT” are used to model the network delay between the global swap manager and the local swap managers. There are entries named “swapdelay” and “reldelay” in these tasks. All of these network delays are set to the mean value of the network delay. The task “serverT” with entry “swap” is used to model the demand of the local swap managers to handle the swapping. The LQN model of the swappable SNMP system is shown in Figure 5-20. The parameters of case one labeled in Figure 5-20 by 1:. The parameters of case two are labeled in Figure 5-20 by 2:.

Figure 5-21. The Swappable SNMP System's LQN Model

In Figure 5-21, the send activity between “sendStart” and “sendEnd” describes that the global swap manager sends out the “SWAP” command, the local swap managers get the command, finish the local swapping, send back the results to the global swap manager and the global swap manager makes decisions through these results. In the LQN model, the global swap manager’s processing is expressed by the join operation in the send activity. The send activity has to wait for all the parallel paths to finish before it is released from the blocked state so that it will always wait for the longest path to finish. The release describes the global swap manager to send out the “OK” command to commit the global swapping.

We have made the following assumptions in the simulation. In case one, when the global swap manager handles the swapping, the SNMP management station also handles the network management by using the polling at the same time. The SNMP management station has to process the polling replies from the local swap managers. All of these actions cause more swapping and preparation delay in case one. So we assume the swapping time in the swappable SNMP agent is 70 ms in case one, and 30 ms in case two. For the actual swapping, the local swapping process time in the swappable SNMP agent is varied and may be less or more than the assumed time. We assume the p_1 , p_2 of “proc” in both cases include the network delay. The network delay d is the mean value of real network delay.

We increased the number of parallel paths in the “send” activity from 2 to 2000 to simulate the increase in the swappable SNMP agents. With the polling and the increase of SNMP agents, the network delay is increased. We simulated the different result with the different network delay from 1ms to 100ms.

5.7.3 Simulation Result

Figure 5-22 shows the comparison between the tested results and the simulated results created by LQN model. The simulation result is very close to the average tested result and this shows that our model simulates the real situation properly.

Figure 5-22. Comparison of Simulated and Tested Global Swapping Time

Figure 5-23. The Simulated Global Swapping Time

By executing the model, we get the result of up to 2000 swappable SNMP agents with different network delay. These simulated results are illustrated in Figure 5-23. In this figure, it is evident that the global swapping time keeps growing linearly according to the increase of the SNMP agents under the same network delay. When the network delay increases, the global swapping time increases too. In the global hot swapping, when all local swap managers reply results to the global swap manager, there will be a queue. This is one reason that affects the swapping time. The other reason is that the global swap manager has to wait for the latest reply from the local swap managers to make the swapping decision, which also affects the swapping time. Because of these reasons, the swapping time increases linearly with the increase of swappable SNMP agents under the same assumption of network delay.

Using the LQN model, we simulated the increasing trend of the swapping time with the swappable SNMP agents under different latency in the network. From the simulation, when the SNMP management station handles the polling, if the mean network delay is 50 ms, the global swapping time is around 152247 ms if there are 2000 swappable SNMP agents. If the mean network delay is 100 ms, the global swapping time is around 272894 ms if there are 2000 swappable SNMP agents. For a network with 2000 managed device, usually each managed device is to be polled every 400 seconds. Compared with 400 seconds, even with 100ms network delay, there should be no trouble carry out the transaction of swapping.

CHAPTER 6. CONCLUSIONS, CONTRIBUTIONS AND FUTURE RESEARCH

6.1 Conclusions

The hot swapping technique is used as an approach in the dynamic updating of software. The infrastructure and its key components are discussed in [4, 5, 6, and 7]. In this thesis, a global swappable SNMP system was studied. Chapter 2 presented the state-of-the-art work in hot swapping. The SNMP framework was presented as well. Chapter 3 described the automatic creation of classes of the S-Module and its S-Proxy from the general Java class. Chapter 4 analyzed the architecture of the swappable SNMP system and the global swapping transaction. Chapter 5 focused on the implementation and application issues related to the hot swappable SNMP system. The scalability of the hot swappable SNMP system is discussed by LQN model that is built using the tested result.

6.2 Contributions

The main contributions of this thesis are as follows:

By using Java reflection, we provide a way to automatically create S-Modules and their corresponding S-Proxy classes from an existing Java class. This provides a useful way to automate the routine aspects of S-Module and S-Proxy creation.

In this thesis, I have implemented the global hot swapping in a distributed computing environment that dynamically updates the distributed modules in the network. No existing approaches have done the similar dynamic updating. A hot swappable SNMP architecture and a hot swappable SNMP system have been constructed in order to implement the global hot swapping process in the distributed environment. The global swapping is constructed by local swappings in a distributed computing environment. The transaction of the global swapping is discussed. Two different kinds of swap managers have been built according to their different functionalities in the global swapping. A set of commands for the global hot swapping has been constructed so that the whole process of swapping could be handled by using these commands. The hot swappable SNMP system allows for a global swapping of the SNMP engine's functions in both the management station and the agents simultaneously while the whole management network continuously provides service. In this thesis, we implement a flexible and dynamically extensible environment for experimentation with a global hot swapping in distributed computing; determine how the global swapping process can be handled in a distributed computing environment; and estimated the global swapping time in swappable SNMP system. A LQN simulation model was built based on the swappable SNMP architecture and the scalability of the swappable SNMP system was discussed.

6.3 Future Research

Hot swapping techniques bring up some profound concepts for software maintenance. This thesis integrated the hot swapping system with SNMP, providing the software dynamic updating and network reconfiguration management in a distributed environment.

Future work can be performed in the several areas. One area that requires further work is the capturing of the S-Module's program state and the mapping of the original S-Module's state to the new S-Module. One approach is extending the JVM [20]. Extending JVM can make program migration, but there is still no approach for extending the JVM and get the check point to store and map the program state between different modules [23, 25, 26, 28, and 29].

Solutions relating to the security of hot swapping techniques are needed. In this hot swapping application, the security is a very important field since we must make sure the swapping to be safety. A strong security solution should be developed in order to allow for the hot swapping technique to be used in sensitive distributed applications.

A hot swapping toolkit can be built to provide a number of services, such as the migration facility to transport S-Modules code to the destination; the communication facility for controlling the global swapping; the security facility to provide authentication and data integrity; and a facility to create the S-Module and corresponding S-Proxy.

REFERENCES

- [1] B. P. Lientz, P. Bennet, E. B. Swanson, and E. Burton. "Software maintenance Management". Addison-Wesley, Reading, 1980.
- [2] J. R. McKee. "Maintenance as a function of design". In Proc. 1984 AIPS national Computer Conference, pages 187-193, 1984.
- [3] S. L. Pfleeger. "Software Engineering: The Production of Quality Software". Macmillan Publishing Company, 2 Edition, 1991.
- [4] Feng, N., Gang, A., White, T., and Pagurek, B. Dynamic Evolution of Network Management Software by Software Hot-Swapping. Accepted for publication at the Seventh IFIP/IEEE International Symposium on Integrated Network Management (IM 2001), Seattle, May 14-18, 2001.
- [5] Feng, N., Ao, G., White, T., and Pagurek, B., "Software Hot-swapping Technology Design", Technical Report SCE-99-04, Systems and Computer Engineering, Carleton University, Ottawa, Canada, June 1999.
- [6] N. Feng, "S-Module Design for Software Hot Swapping", Masters Thesis, System and Computer Engineering Department, Carleton University, September 1999

-
- [7] G. Ao, "Software Hot-Swapping Techniques for upgrading Mission Critical Applications On the Fly", Masters Thesis, System and Computer Engineering Department, Carleton University, February 2000
- [8] L. Tan, B.Esfandiari, B. Pagurek, "The SwapBox: A Test Container and a Framework for Hot-Swappable JavaBeans", In Proc. of the WCOP (Workshop on Component-Oriented Programming) workshop (at ECOOP 2001), Budapest, Hungary, June 19, 2001.
- [9] J.Kramer, J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management", IEEE Transactions on Software Engineering, Vol 16, No 11, November 1990
- [10] D.Gupta, P.Jalote, G. Barua. "A formal framework for on-line software version change". IEEE Transactions on Software Engineering, vol 22, no 2, February 1996
- [11] Mark E. Segal, Ophir Frieder, "On-the-Fly Program Modification: Systems for Dynamic Updating", IEEE Software, March/April 1993 (Vol. 10, No. 2)
- [12] William Stallings, "SNMP, SNMPv2, SNMPv3, and RMON 1 and 2", third edition Addison-Wesley, 1999

-
- [13] J. Case, R. Mundy, D. Partain, B. Stewart, "Introduction to SNMPv3", RFC2570, April 1999
- [14] Harrington, D., Presuhn, R. and B. Wijnen, "An Architecture for describing SNMP Management Frameworks", RFC 2571 draft , April 1999
- [15] J. Case, D. Harrington, R. Presuhn, B. Wijnen "Message Processing and Dispatching" RFC 2572 May 1999
- [16] D.Levi, P.Meyer, B.Stewart, "SNMPv3 Application", RFC2573, April 1999
- [17] U.Blumenthal, B.Wijnen, "The User-Based Security Model for Version 3 of the Simple Network Management Protocol (SNMPv3)", RFC2574, April 1999
- [18] G. Waters, "User-based Security Model for SNMPv2, RFC1910, February 1996
- [19] O. Cherkaoui , Ylian Saint Hillaire, Ahmed Sehrhouchni, Hafedh Mili, "Towards a modular and interoperable SNMPv3", IEEE 1998 Enterprise Networking and Computing Conference, Atlanta, GA, 1998
- [20] B. Venners, "Inside the Java Virtual Machine", 1998, the McGraw-Hill Companies, Inc.

-
- [21] “Java Object Serialization Specification” 1997-1999, Sun Microsystems, Inc.
<http://java.sun.com/products/jdk/1.1/docs/guide/serialization/spec/serialTOC.doc.html>
- [22] “JavaTM Core Reflection” February 4, 1997, Sun Microsystems, Inc.
<http://java.sun.com/j2se/1.3/docs/guide/reflection/spec/java-reflection.doc.html>
- [23] Tim Lindholm, Frank Yellin, “ The JavaTM Virtual Machine Specification”,
September 1996, Sun Microsystems, Inc.
- [24] H. Gomaa. “Designing Concurrent, Distributed, and Real-Time Application with UML”. Addison-Wesley, 2000.
- [25] James S. Plank, “An overview of Checkpointing in Uniprocessor and Distributed Systems, Focusing on Implementation and Performance”, Technical Report, July, 1997
- [26] Yi-Min Wang, Yennun Huang, Kiem-Phong Vo, “Checkpointing and Its Application”, Proc. IEEE Fault-tolerant Computing Symp. (FTCS-25), pp22-31, June 1995
- [27] Philip A. Bernstein, Eric NewComer, “Principles of Transaction Processing”, 1997.
Morgan Kaufmann Publishers, Inc.

-
- [28] R. Riggs, J. Waldo, A. Wollrath, K. Bharat, "Pickling State in the Java System", USENIX Conference on Object-Oriented Technologies (COOTS), Canada, 1996
- [29] S. Bouchenak and D. Hagimont. "Approaches to capturing Java Thread State", Middleware 2000, poster session
- [30] Paul Hyde , "JavaTM Thread Programming", Sams, March 2001
- [31] Truyen, E., Jorgensen, B. N., Joosen, W., Verbaeten, P.: "On Interaction Refinement in Middleware", Presented at WCOP 2000 (Sophia Antipolis, France, June 2000)
- [32] De Palma N., Bellissard L., Riveill M. "Dynamic Reconfiguration of Agent-Based Application", Proceedings of the 3rd European Research Seminar on Advances in Distributed Systems (ERSADS'99), Madeira Island, Portugal, 23-28 April 1999.
- [33] Peyman Oreizy, Nenad Medvidovic and Richard N. Taylor, "Architecture-Based Runtime Software Evolution", The 20th International Conference on Software Engineering, 1998 (ICSE98)
- [34] Plasil, F., Balek, D., Janecek, R., "SOFA/DCUP: Architecture for Component Trading and Dynamic Updating", Proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press, May 1998

-
- [35] Murray Woodside, “Layered Performance Modeling and Layered Queueing: Quick Tutorial”, <http://www.sce.carleton.ca/courses/94511/handouts/lqns-tutorial.pdf>