

S-Module Design for Software Hot-swapping

By

Ning FENG

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

Ottawa-Carleton Institute for Electrical and Computer Engineering
Faculty of Engineering
Department of Systems and Computer Engineering
Carleton University
1125 Colonel Drive
Ottawa, Ontario, Canada
K1S 5B6

November 25, 1999

Copyright
1999, Ning FENG

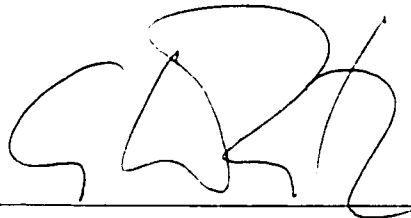
The undersigned hereby recommend to
the faculty of Graduate Studies and Research
acceptance of the thesis

S-Module Design for Software Hot-swapping

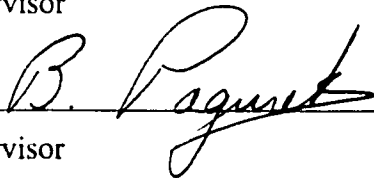
submitted by

Ning FENG

in partial fulfillment of the requirements for the degree of Master of Engineering



Thesis Co-supervisor



Thesis Co-supervisor



Chair, Department of System and Computer Engineering

Carleton University
November 25, 1999

Abstract

Research on software hot-swapping technology is driven by the increasing demands for on-line software upgrading without the interruption of system service. This thesis proposes a new hot-swapping infrastructure for designing and implementing software applications.

Several software design patterns have been investigated and analyzed for the design of the atomic swappable unit, the S-Module. The Proxy Pattern has been promoted to a detailed design, which leads to a development of the hot-swapping framework.

In the network management area, the SNMP world has recognized the value of dynamic upgrading and extension in recent years. The modular structure of SNMPv3 also provides a good test-bed for our new hot-swapping framework. A real hot-swapping application based on the SNMPv3 agent has demonstrated that the goal of hot-swapping can be achieved by following the technology and methodology described in this thesis.

Acknowledgments

First, I wish to express my profound gratitude to my supervisor, Professor Bernie Pagurek, for his consistent guidance and support during the course of my research. I would like to express my special thanks to my co-supervisor, Mr. Tony White, for his invaluable guidance, inspiration, and encouragement throughout the span of this research. Without their supervisions, my work would not be of the same quality.

My thanks also go to colleague Gang Ao, who provided tremendous background research in the area which this thesis is based upon.

I also wish to acknowledge the research funding from the Nortel Networks.

Finally, I would like to express my deep appreciation to my husband Jingdong, my son Kevin, and my parents, without whose love, encouragement, and support this work would not have been possible.

Table of Contents

Chapter 1	Introduction.....	1
1.1	Background.....	1
1.2	Objectives	2
1.3	Organization.....	3
Chapter 2	Software Hot-Swapping Infrastructure.....	5
2.1	Introduction.....	5
2.2	New Software Hot-swapping Infrastructure	6
2.2.1	Terms.....	6
2.2.2	Two Types of Software Hot-Swapping.....	6
2.2.3	Research on Software Hot-Swapping Technique	11
Chapter 3	Several Approaches in Designing S-Modules	13
3.1	S-Module	13
3.1.1	Characteristics of an S-Module.....	14
3.1.2	Life Cycle of an S-Module	15
3.2	The JVM Modification Approach.....	16
3.2.1	JVM	16
3.2.2	Hot-swapping in JVM.....	17
3.2.3	Pros and Cons	18
3.3	The Observer Pattern Approach	19
3.3.1	Observer Pattern	19

3.3.2	Hot-swapping by Using the Observer Pattern	21
3.3.3	Pros and Cons	26
3.4	The Proxy Pattern Approach.....	27
3.4.1	Proxy Pattern.....	27
3.4.2	Hot-swapping by Using the Proxy Pattern	28
3.4.3	Pros and Cons	31
3.5	The Mediator Pattern Approach	32
3.5.1	Mediator Pattern	32
3.5.2	Hot-swapping by Using the Mediator Pattern	33
3.5.3	Pros and Cons	36
3.6	Conclusions.....	37
Chapter 4	Designing S-Modules by Using Proxy Pattern	39
4.1	Logical View	40
4.2	Rules for Designing an S-Module	43
4.2.1	Identity	44
4.2.2	Service	46
4.2.3	Internal State	49
4.2.4	Dependency List	52
4.2.5	Mapping Rule and Persistency	54
4.3	Rules for Designing an S-Proxy	57
4.3.1	Attributes	58
4.3.2	Interfaces for Clients.....	59
4.3.3	Interfaces for the Swap Manager	61

4.4	Major Abstract Classes Definition.....	63
4.4.1	S_ModuleSuper Class.....	63
4.4.2	S_ProxySuper Class.....	65
4.4.3	S_Main Class.....	65
4.4.4	S_ModuleTypeX.....	66
4.4.5	S-ProxyTypeX.....	66
4.5	Component Diagram.....	66
4.6	Sequence Diagram.....	67
4.7	Scenarios.....	69
4.7.1	New and Old S-Modules Have Same Behavior Methods.....	69
4.7.2	New S-Module Does Not Have All Old S-Module Methods.....	71
4.7.3	New S-Module Has Methods That Old S-Module Does Not.....	73
Chapter 5	Hot-swapping Application: A Swappable SNMP Agent.....	75
5.1	SNMP Architecture.....	76
5.1.1	Background Information.....	76
5.1.2	SNMP Architecture.....	77
5.1.3	User-Based Security Model.....	79
5.1.4	Abstract Service Interface.....	80
5.2	Swappable SNMP Agent.....	81
5.2.1	System Architecture.....	81
5.2.2	S-Components.....	83
5.2.3	Class Diagram.....	85
5.3	Tests and Results.....	87

Chapter 6	Conclusions, Contributions and Suggestions for Future Research.....	92
6.1	Conclusions.....	92
6.2	Contributions	93
6.3	Suggestions for Future Research	93
References	96

Table of Figures

FIGURE 2-1	Software Hot-swapping at the Program Level.....	7
FIGURE 2-2	Software Hot-swapping at the Module Level	9
FIGURE 3-1	The JVM Modification Approach	18
FIGURE 3-2	Observer Pattern Structure	20
FIGURE 3-3	Observer Pattern Collaboration Diagram	21
FIGURE 3-4	The Observer Pattern Approach	22
FIGURE 3-5	Object Reference Propagation Problem	24
FIGURE 3-6	Proxy Pattern Structure	27
FIGURE 3-7	Proxy Pattern Object Diagram	28
FIGURE 3-8	The Proxy Pattern Approach	29
FIGURE 3-9	Mediator Pattern Structure	32
FIGURE 3-10	Mediator Pattern Object Diagram	33
FIGURE 3-11	The Mediator Pattern Approach	34
FIGURE 4-1	Designing S-Module by Using Proxy Pattern	39
FIGURE 4-2	Logical View of Programs with S-Modules	41
FIGURE 4-3	State Diagram	51
FIGURE 4-4	Dependency Map and Deadlock Condition	53
FIGURE 4-5	Interfaces In an S-Module and Its S-Proxy	67
FIGURE 4-6	Sequence Diagram	68
FIGURE 4-7	New and Old S-Module Have Same Behavior Methods	70
FIGURE 4-8	New S-Module Does Not Have All Old S-Module Methods	72
FIGURE 4-9	New S-Module Has Methods That Old S-Module Does Not	74

FIGURE 5-1	Command Responder	80
FIGURE 5-2	A Swappable SNMP Agent	83
FIGURE 5-3	Swappable SNMP Agent Class Diagram (User-Based Security Model).....	86
FIGURE 5-4	Test System Environment	87

Chapter 1 Introduction

1.1 Background

Change is inevitable in software. Software changes such as bug-fixes, updates, or functionality upgrades are generally referred as software upgrading. It has been an issue since the running of the first software program in the computer world. A hot research topic in software upgrading is the on-line real-time software upgrading which meets the requirement of not interrupting system services. It is very crucial under certain circumstances, i.e. in telecommunication networks and real-time control system, where the interrupting of services for software upgrading is unacceptable. Software upgrading is generally not easy, especially in computer communication networks where software can be widely distributed across heterogeneous domains. The capability to provide the continuity of service in a computer network, or software applications in general, has become an important asset for the service provider to stay ahead of the competitors.

Studies on software upgrading techniques have shown that current deployed technologies are not well suited for a smooth transition of services due to upgrades and bug-fixes [1]. Most of these technologies, such as the code patch, require the server, in which the application resides, to be shut down and the system to be re-booted. Since software upgrading can't be avoided, and in some cases must be accomplished on-line, software upgrading should be planned and prepared at design time. A new solution is highly demanded.

Research on software hot-swapping technology is driven by increasing demand for on-line real-time software upgrading without the interruption of system services. The term “hot-swapping” comes from the hardware world which normally means the replacement of a hardware module such as a CD-ROM driver, power supply, or other devices with a similar device while the computer system using them remains in operation. Similarly, software hot-swapping refers to the replacement of a software program or part of a program while the whole software system remains in service.

The development in computer technology and software development methodology makes it possible to achieve the software hot-swapping. For example, the form and acceptance of software design patterns paves the road for standard software design and implementation which in turn makes software upgrading easier. Software design patterns offer a method of documenting proven designs and evaluating aspects that can vary [3]. A design pattern addresses a recurring problem in a given context and a proven solution to that problem.

In this thesis, a new infrastructure for a software hot-swapping system is proposed and the components of the infrastructure are introduced. The thesis focuses on the research on the design of the atomic unit of software swappability, the swappable module (S-Module), by applying different design patterns. A detailed design based on the Proxy Pattern approach is developed and the design has been applied to a real application, the Simple Network Management Protocol (SNMP) version 3 agent, for the experimental purpose.

1.2 Objectives

The main objectives of this thesis are as follows:

1. Propose a new software design/implementation infrastructure for software hot-swapping application programs based on the object-oriented programming and distributed computing techniques.
2. Conduct research for defining the atomic software unit of the hot-swapping system. Propose and analyze several approaches based on different design patterns and recommend on the design choice. Define the designing rules for the S-Module in the hot-swapping framework by using the recommended approach.
3. Apply the new hot-swapping framework to a real application as a test-bed. Demonstrate and test the design of the hot-swapping framework.

1.3 Organization

The remaining chapters of this thesis are organized as follows:

Chapter 2 briefly overviews the current techniques of software upgrading, and proposes a new infrastructure for software on-line upgrading. The research target of this thesis is high-lighted again at the end of this chapter.

Chapter 3 describes and analyzes four potential approaches in designing the swappable software module, which is called the S-Module. These four approaches are: a Java Virtual Machine modification approach, an Observer Pattern approach, a Proxy Pattern approach, and a Mediator Pattern approach. After comparative analysis of the advantages and the disadvantages, the Proxy Pattern approach is selected as the most promising one.

Chapter 4 gives a detailed design of the hot-swapping framework based on the Proxy Pattern approach. The focus of the design is on the characteristics of the S-Module, the S-Proxy, and their relationship with the Swap Manager.

Chapter 5 presents how a real application in the network management area is designed and reconstructed following the rules defined in Chapter 4, thus making it a hot-swapping application. The application, a swappable SNMP agent, is first introduced, followed by a description of the class hierarchy. Test results and limitations are summarized at the end of this chapter.

In Chapter 6, this thesis is concluded, contributions and recommendations for future research are discussed.

Chapter 2 Software Hot-Swapping Infrastructure

2.1 Introduction

The research on software hot-swapping technologies evolves along the development of object-oriented (OO) programming/system. Early research on software hot-swapping, such as Common Lisp Object System (CLOS) [1], focuses on dealing with the issues such as dynamic typing, run-time interpretation, etc. which in turn are adopted by the next generation OO programming languages such as Java. The new OO languages (Java, C++, etc.) and programming environments enable us to take full advantage of OO methodology and its capabilities for creating flexible, modular, and swappable software programs.

Software hot-swapping technology is also affected by the development in distributed computing technology, such as Javabeans, RMI etc. [1]. The ubiquitous client-server architecture broadens the scope of software hot-swapping from a stand alone system to a group of interconnected systems.

Developments in OO programming and distributed computing technologies bring us both challenges and opportunities to develop a new software hot-swapping infrastructure. The purpose of this thesis is to analyze this infrastructure, its components, and key algorithms.

In the next section, some terms are introduced first, continued by the definition of two types of software hot-swapping. The hot-swapping infrastructure and its components that this thesis focuses on is described at the end.

2.2 New Software Hot-swapping Infrastructure

2.2.1 Terms

The research subjects in software hot-swapping are software **programs** and **modules**. In the context of this thesis, a software **program** is an executable software entity which is made up of one or more software modules. A software **module** has certain functions which provide services to other modules in the program. Through the interactions between modules, a program provides certain services to other programs, which in turn forms a complex software system.

In the OO world, everything is an object. We define a module as an object which is made up of other objects. The services provided by the module are defined in the behavior of the objects.

2.2.2 Two Types of Software Hot-Swapping

Based on the definitions about software program and software module, we can have two types of software hot-swapping: **program level** software hot-swapping and **module level** software hot-swapping.

2.2.2.1 Program Level Software Hot-swapping

At this level, the swappable software entity is a program within a complex software system. The on-line upgrading of a software program within this system should not interrupt the system services. We define such a system as a hot-swapping system. The swappable program is called an S-Program in contrast to non-swappable programs. In order to conduct the program swapping, a mediator or a controller is needed, we call it an S-Program-Swap-Manager, which controls the hot-swapping transaction between S-Programs, and interacts with other programs. For example, a hot-swapping system, as shown in Figure 2-1, is composed of one S-Program-Swap-Manager program, several S-Programs and non S-Programs, as well as some client programs which are software entities that consume the services provided by the S-Programs.

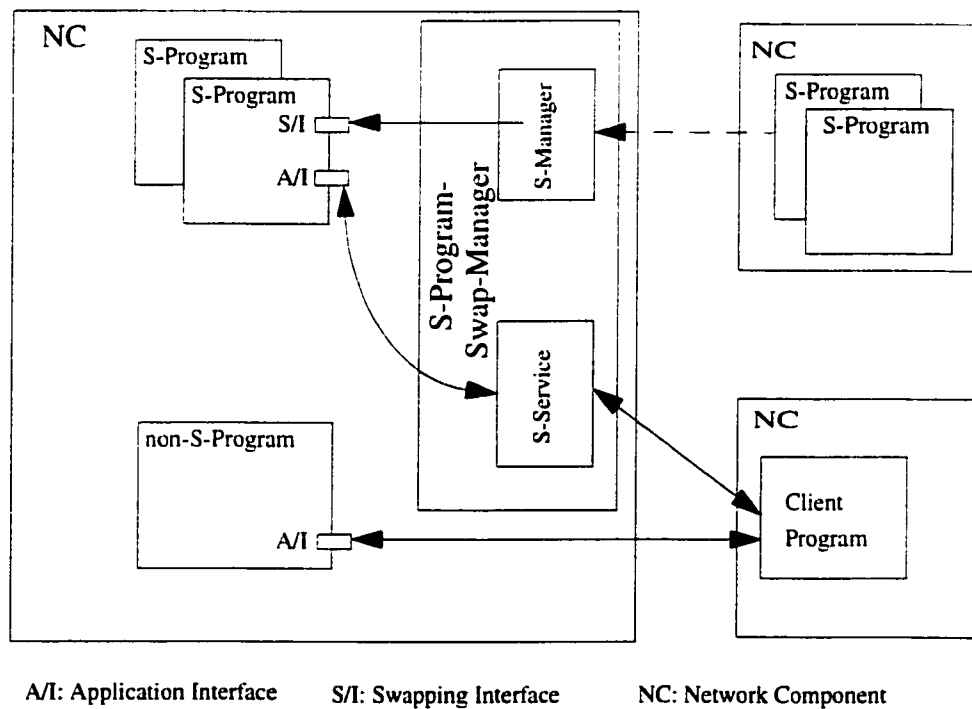


FIGURE 2-1. Software Hot-swapping at the Program Level

The S-Program-Swap-Manager is the control centre of the whole hot-swapping system. It can access and control all the S-Programs on the same network component. All client programs must contact the S-Program-Swap-Manager in order to get the services of S-Programs. The S-Program-Swap-Manager has two parts: the S-Manager and the S-Service. The S-Manager is in charge of the hot-swapping transaction between two S-Programs, the old version S-Program and the new version S-Program. The S-Service acts as an interface for client programs on the other network components to use the services of S-Programs. However, client programs can use services of non-S-Programs directly, without interacting with the S-Service.

Each S-Program should provide two interfaces: a swapping interface (S/I in the Figure 2-1) for interacting with the S-Manager, and an application interface (A/I) for the S-Service. The non-S-Programs provide only an application interface to communicate with the client programs (Figure 2-1).

The hot-swapping transaction happens only between the S-Program-Swap-Manager and different versions of the same S-Program. It is transparent to other S-Programs, client programs and non-S-Programs.

2.2.2.2 Module Level Software Hot-swapping

At this level, the swappable software entity is a module within an application program. Replacing or upgrading the swappable modules during the execution of an application will not interrupt the services of the program. We define such an application as a hot-swapping application called S-Application. The swappable module is called an S-Module in contrast to non-swappable modules.

In order to conduct S-Module swapping, a mediator is needed. We call it the Swap Manager, which controls the hot-swapping transaction. Figure 2-2 shows an S-Application composed of one Swap Manager, several S-Modules and non-S-Modules.

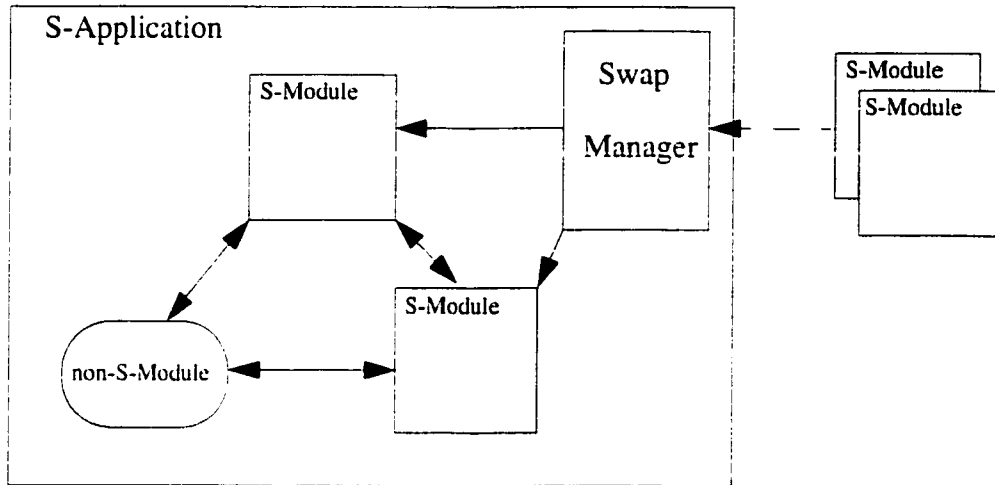


FIGURE 2-2. Software Hot-swapping at the Module Level

An S-Module exists within an S-Application program, it is not a stand alone executable entity. An S-Module cooperates with other S-Modules and non-S-Module components inside the same application program, provides certain services during run-time, and communicates with the Swap Manager.

The Swap Manager has access to all S-Modules. As defined by Gang Ao in an initial study, it provides the following services [1]:

- **Listening service**, which waits for the arrival of new S-Modules and instantiates them.
- **Security service**, which performs the authentication checking of the incoming S-Module.
- **Transaction service**, which provides the control of hot-swapping transaction.

- **Timing service**, which ensures timely completion of the hot-swapping transaction.
- **Event service**, which informs the relevant entities about the hot-swapping actions.
- **Repository service**, which caches the states of the old S-Module during the hot-swapping transaction.

A basic scenario of the module level hot-swapping is as follows:

1. An S-Application is running and in service. One or a group of new S-Modules are sent by an administrator to the S-Application.
2. The listening service of the Swap Manager in the S-Application receives the codes (new S-Modules) and checks the authentication of the loaded codes.
3. If the incoming codes pass the security check, the Swap Manager instantiates corresponding objects, and checks if the new S-Modules have dependent S-Modules which must be in the system before this S-Module could be swapped in.
4. The Swap Manager targets those S-Modules to be swapped out, blocks new calls to them and starts timing the hot-swapping transaction.
5. When all the targeted S-Modules are ready to be swapped out, the Swap Manager gets and stores all the necessary attributes of these S-Modules and mapped the attributes to the new S-Modules.
6. After the attributes mapping, all new S-Modules are swapped in replacing the old S-Modules, and the blocked calls are released. The timing service stops timing. The hot-swapping transaction is completed successfully.

7. If within a certain time limits, any of the old S-Modules is not ready to be swapped out or any of the new S-Modules fails the attributes mapping, the whole hot-swapping transaction aborts. The blocked calls on the old S-Modules are released by the Swap Manager and the old S-Modules are resumed.

2.2.3 Research on Software Hot-Swapping Technique

Software hot-swapping is a complex issue involving multiple technologies, such as OO technology, distributed computing technology, programming language technologies, etc. As at the initial stage of the research on hot-swapping technique, this thesis only focuses on the hot-swapping at the software module level.

One of the main issues in module level hot-swapping is the design of S-Modules. Other issues include defining a protocol for mobile S-Modules (code), loading codes and instantiating objects, as well as defining the services provided by the Swap Manager. This thesis discusses and analyzes some possible approaches for designing S-Modules in a hot-swapping application, and gives suggestions on the design choice. A detailed study on the characteristic of the Swap Manager is currently under research by colleague Gang Ao [2].

In the next chapter we will introduce four possible approaches for designing an S-Module: a Java Virtual Machine (JVM) modification approach, an Observer Pattern approach, a Proxy Pattern approach, and a Mediator Pattern approach. The last three approaches are based on design patterns [3]. A design pattern addresses a recurring design problem that arises in specific situations and describes a solution to it. For these approaches, first the design pattern is introduced, then how to apply the particular pattern in the hot-swapping application is discussed, and finally an analysis of this approach is given.

All the approaches are based on the following assumptions:

- The software is written in Java, taking the advantage of Java's features, such as object-oriented, distributed, interpreted, robust, secure, architecture-neutral, portable, persistent, multi-threaded, as well as providing dynamic type checking and automatic garbage collection.
- An S-Module includes one object or a group of objects. We are talking about hot-swapping at the object (instance of a class) level.

It is worth mentioning here that because Java is a strongly typed language, if two methods have the same syntax (method name and argument lists) then the behaviors of the methods are not distinguishable.

Chapter 3 Several Approaches in Designing S-Modules

3.1 S-Module

Before we go ahead with the discussion of the design of an S-Module, it is necessary to investigate and identify the characteristics of an S-Module.

As mentioned in the previous chapter, an S-Module is a software entity which can be replaced or upgraded during the execution of a program. The biggest difference between an S-Module and a non-S-Module is that upgrading an S-Module will not interrupt the running of the program, while upgrading a non-S-Module generally needs re-compilation, re-linking and re-starting of the program. An S-Module exists within a program, it is not a stand alone entity, instead it interacts/cooperates with other S-Modules and non S-Modules within the same application program. The special requirements and existing environment decide that an S-Module should have a set of specific characteristics in contrast to a non-S-Module. When we design an S-Module, those characteristics have to be taken into consideration. Another aspect we need to consider in the S-Module design is its life cycle which defines a procedure that an S-Module will follow. The life cycle of an S-Module will be discussed in Section 3.1.2 .

3.1.1 Characteristics of an S-Module

- **Identity:** An S-Module has a unique existence and identity which can be used to identify an S-Module of a certain type in the scope of a certain swapping model. The identity of an S-Module also provides detailed information about the S-Module such as a version with major and minor indices and annotation that describes the nature of the changes from a previous version, the name of the vendor and the date of release.
- **Service:** An S-Module should provide a set of services. Same type of S-Modules should have the same set of services. A service specification provides the guideline for the services. For example, a security S-Module should provide the following basic services: encryption, decryption, authorization, and authentication. If two S-Modules are to be swappable, they must at least provide the same services.
- **Internal State:** An S-Module has a set of internal states. In some states an S-Module is swappable, in others not. If and only if an S-Module is in its swappable state, can a hot-swapping transaction take place. Any services provided by an S-Module should be finished within a finite time, and the S-Module returns to the “idle” state after completing the service.
- **Dependency List:** An S-Module should have a dependency list which includes the S-Modules that this S-Module depends on. Before an S-Module can be activated, the S-Modules that it depends on must have been loaded into the S-Application and been activated.
- **Mapping Rule:** Associated with each S-Module is a set of mapping rules which control the transfer of the attributes between versions of the S-Module.

- **Persistence:** During a hot-swapping transaction, the system resources held by an S-Module, such as opened files, in-service communication channels, etc., should be released or transferred to the new S-Module.
- **Administration Interface:** An S-Module should provide an interface for the Swap Manager to manage it.

3.1.2 Life Cycle of an S-Module

The following is a description of the life cycle of an S-Module:

1. A new S-Module is defined and designed according to a **service** specification.
2. The new S-Module is assigned a unique **identity**.
3. The new S-Module is sent to the Swap Manager of a running S-Application, and the Swap Manager identifies the “old” S-Module that is to be replaced.
4. The Swap Manager will check the internal **state** of the “old” S-Module through its **administration interface**. If the S-Module is in a swappable state, a swapping transaction will happen and the S-Module enters the “swapping state”.
5. By applying the **mapping rules**, the internal states (attributes) of the “old” S-Module will be converted to the internal states of the new S-Module.
6. The system resources allocated by the “old” S-Module will be released or transferred to the new S-Module to achieve **persistency**.

7. The Swap Manager activates the new S-Module to provide services until it is replaced by another S-Module. The “old” S-Module will be removed from the program by the Swap Manager after being swapped successfully.

3.2 The JVM Modification Approach

3.2.1 JVM

The Java Virtual Machine (JVM) is an abstract computer that runs compiled Java programs. The JVM is “virtual” because it is generally implemented in software on top of a “real” hardware platform and operating system. The JVM is central to Java’s portability because compiled Java programs run on the JVM, independent of whatever may be underneath a particular JVM implementation.

The Java Virtual Machine can be divided into five fundamental pieces: a byte-code instruction set, a set of registers, a stack, a garbage-collected heap, and an area for storing methods. These parts are abstract but they must exist in some form in every JVM implementation.

The heap is the run-time data area from which memory for all class instances and arrays is allocated. The Java heap is created on virtual machine start-up. Heap storage for objects is reclaimed by an automatic storage management system (typically a garbage collector); objects are never explicitly de-allocated [4].

Java is a strongly typed language, which means that every variable and every expression has a type that is known at compile time. The types of the Java language are divided into two categories: primitive types and reference types. A primitive type is a type that is pre-defined by the Java language and named by a reserved keyword. A reference type is one of the three kinds: the class types, the interface types and the array types. An object is a dynamically created class instance or an array. The reference values (often just *references*) are *pointers* to these objects and a special null reference, which refers to no object.

Java objects are referenced indirectly at run-time, via *objectref*, which is a kind of indirect pointer into the heap. Objects are automatically garbage-collected in Java, the programmers do not have to (and, in fact, can not) manually free the memory allocated to an object when they are finished using it. Because objects are never referenced directly, parallel garbage collectors can be written that operate independently from the program, moving around objects in the heap [5].

3.2.2 Hot-swapping in JVM

The Java Virtual Machine specification [4] does not require any particular internal structure for objects. Different JVM implementations may use different techniques. In Sun's current implementation of the Java Virtual Machine, a reference to a class instance is a pointer to a handle that is itself a pair of pointers: one to a table containing the methods of the object and a pointer to the *Class* object that represents the type of the object, and the other to the memory allocated from the Java heap for the object data.

Figure 3-1 shows the indirect referencing of the JVM. *object1*, *object2* and *object3* are all allocated on the heap. *object2* and *object3* are objects of the same type. *object1* has a reference to *object2*, this is implemented by *object1* holding a pointer to a handle which again has a pointer to the address where *object2* is. This indirect referencing not only allows a garbage collector to operate independently from the program, but also give us a chance to hot-swap one object with another at run-time. If the pointer to *object2* in the handle of *object1* can be changed with a pointer to another object *object3*, together with a proper state mapping process, then *object2* is hot-swapped with *object3*. *object1* will not be aware of the change.

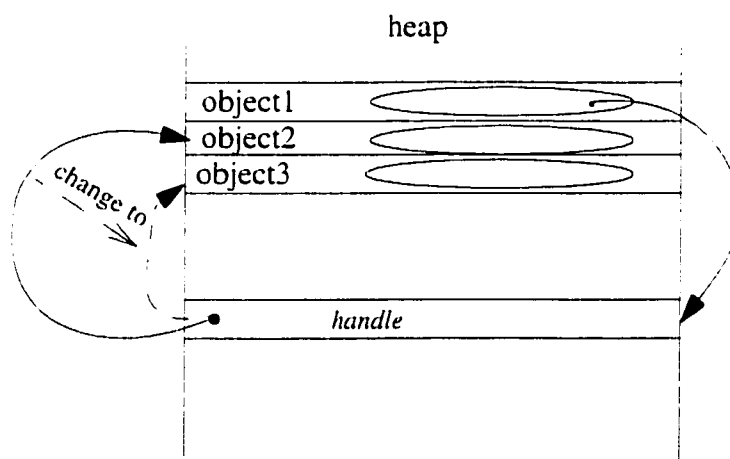


FIGURE 3-1. The JVM Modification Approach

3.2.3 Pros and Cons

This idea of hot-swapping objects inside the JVM is straight forward but to implement it is not as simple as it may seem. An object reference in Java is actually an indirect index to the real pointer, and a large object table exists to map these indirect indexes into the actual

object reference [5]. This indirection allows the garbage collector to mark, sweep, move, or exam one object at a time. Each object can be independently moved out from a running Java program by changing only the object table entries. Running in a separate thread, Java's parallel garbage collector cleans up the Java environment silently and in the background.

Different implementation of Java Virtual Machine use different techniques, so the design of hot-swapping system is JVM specific.

To write a hot-swapping system inside the JVM, all these factors must be considered and be taken care of. The difficult part could be collaborating with the garbage collector, which is already a complicated algorithm. A "smart" solution would probably involve writing a swap-aware garbage collector. On the other hand having the JVM take care of the hot-swapping transaction makes the application program easy to write.

Changing or modifying the JVM, however, will introduce an extension to the Java standard which is not the interest of our research.

3.3 The Observer Pattern Approach

3.3.1 Observer Pattern

The Observer Pattern defines a one-to-many dependency between objects so that when one object changes state, all its dependents are notified and updated automatically [3].

The key objects in this pattern are *subject* and *observer*. A *subject* may have any number of dependent *observers*. All *observers* are notified whenever the *subject* undergoes a change in state. In response, each *observer* will query the *subject* to synchronize its state with the *subject's* state.

An Observer Pattern is typically represented as the following class diagram (Figure 3-2).

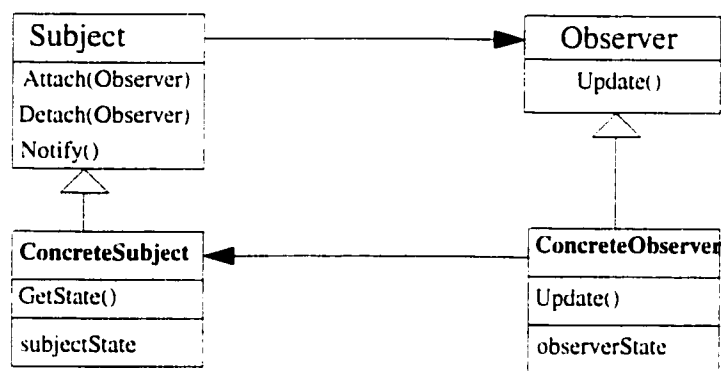


FIGURE 3-2. Observer Pattern Structure

Subject: maintains a list of *observers* and provides an interface to attach and detach *observers* at the runtime.

Observer: provides an update interface to receive signal from *subject*.

ConcreteSubject: stores *subject* state interested by *observers* and sends notification to its *observers* when any change occurs.

ConcreteObserver: maintains *observer* state and a reference to a *ConcreteSubject* object; implements update interface to keep its state consistent with the *subject's*.

An Observer Pattern collaboration diagram is shown in Figure 3-3:

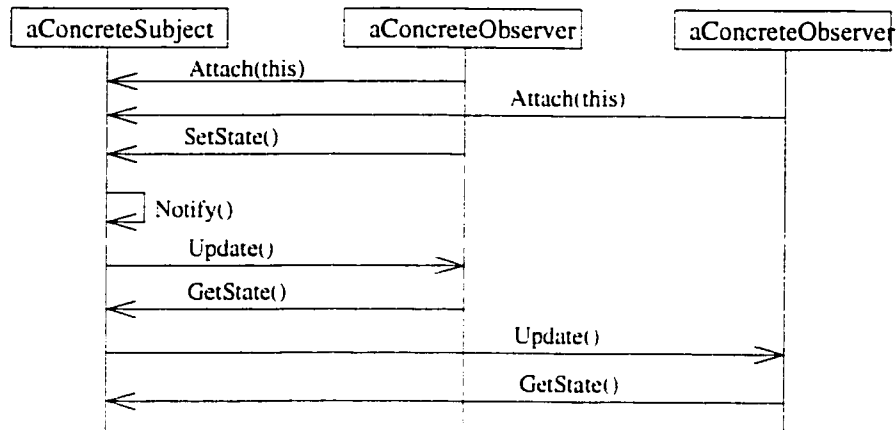


FIGURE 3-3. Observer Pattern Collaboration Diagram

3.3.2 Hot-swapping by Using the Observer Pattern

3.3.2.1 Rationale

One of the challenging issues in designing a software hot-swapping system is to solve the object reference problem. This approach comes from the idea that when an old S-Module is hot-swapped with a new one, all the objects that have a reference to the old S-Module should be notified and thus be forced to update the reference to point to the new S-Module.

As shown in Figure 3-4, a hot-swapping program includes a Swap Manager, several S-Modules, a subject (*S-ModuleSubject*) which takes the S-Modules as its internal states, and some non-S-Modules. The relationship between *S-ModuleSubject* and S-Module can be 1:1 or 1:many.

Any object (S-Module or non-S-Module) that interacts with an S-Module is an observer of that S-Module. The subject that the observers register for, such as the *S-ModuleSubject* in the figure, is also a container object which takes all the S-Modules as its internal states. Any object (*observer1*) that wants to interact with an S-Module, such as *oldS-Module1*, must register itself with the *S-ModuleSubject* as an observer of a state (*oldS-Module1*), and get the reference to that S-Module so that it can invoke methods of the *oldModule1* directly. When the *oldS-Module1* is hot-swapped with the *newS-Module1*, all the registered observers (*observer1*) of the *oldS-Module1* will be notified and updated with the reference to the *newS-Module1*.

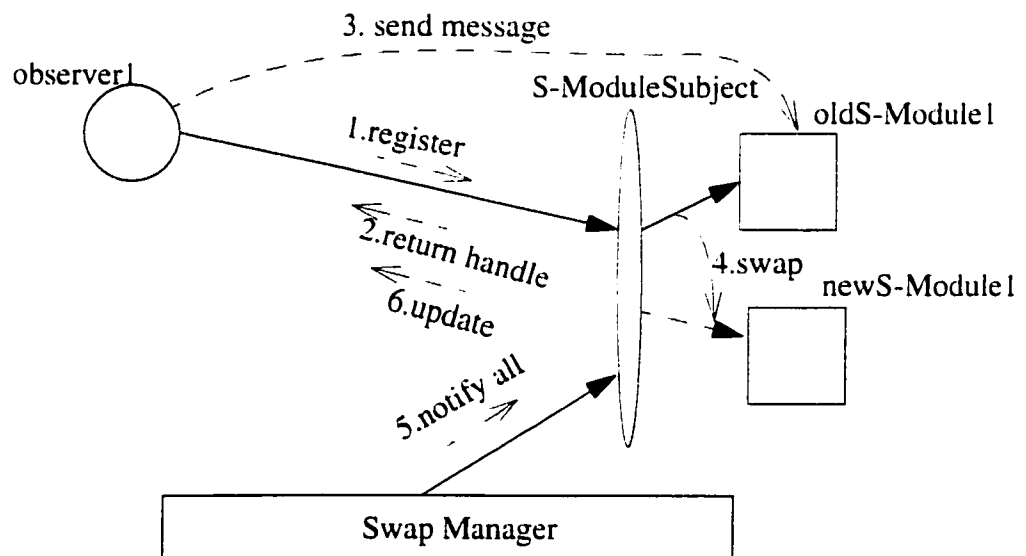


FIGURE 3-4. The Observer Pattern Approach

The subject (*S-ModuleSubject*) will handle the administration of the observers as well as manage and maintain its states (S-Modules). The *Swap Manager* will contact the *S-ModuleSubject* to control the swap transaction.

The sequence of messaging is as follows:

1. An object (*observer1*) wants to access the services provided by an old S-Module (*oldS-Module1*), it must first contact the corresponding S-ModuleSubject and register itself as an observer of the state (*oldS-Module1*).
2. After registration, the *S-ModuleSubject* will pass the handle of the *oldS-Module1* to *observer1*. From now on the *observer1* has a reference to the *oldS-Module1*.
3. Using the acquired object reference, *observer1* requests S-Module's services by sending messages to *oldS-Module1* directly.
4. Now the *Swap Manager* wants to swap the *oldS-Module1* with a new S-Module (*newS-Module1*). Making sure that the *oldS-Module1* is in a swappable state, for example, no other objects are currently sending messages to the *oldS-Module1*, the *Swap Manager* retrieves the states of the *oldS-Module1* and converts the states to the *newS-Module1* properly according to the mapping rules, then swaps the *oldS-Module1* with the *newS-Module1*.
5. After swapping the *oldS-Module1* with the *newS-Module1*, the *Swap Manager* sends a notification message to the *S-ModuleSubject* and asks all the related registered observers be notified of the change.
6. The *S-ModuleSubject* will then send messages to the related observers to update their references to refer to the *newS-Module1*.
7. From now on, *observer1* can send messages to the *newS-Module1* directly. Since no reference to the *oldS-Module1* exists any more, the *oldS-Module1* can be garbage collected later by the Java Virtual Machine.

3.3.2.2 Known Issues

3.3.2.2.1 Reference Propagation Issue

This approach works well with direct referencing like the *observer1* in Figure 3-4. There are cases when other objects get a handle of the *oldS-Module1* from the *observer1* but not from the *S-ModuleSubject* directly. This indirect referencing will propagate the handle of the *oldS-Module1* to other objects which have not registered with the *S-ModuleSubject*, thus those objects will not be notified when the *oldS-Module1* is swapped with the *newS-Module1*. The approach for hot-swapping will fail in this case.

Figure 3-5 shows the propagation problem. An object (*observer1*) has acquired a direct handle of the S-Module (*aS-Module*) by registering itself with the subject (*S-ModuleSubject*). Another object (*anObject*) has not registered directly with the *S-ModuleSubject* as an observer of *aS-Module*, however when *anObject* sends a message to *observer1*, *observer1* could pass the reference of *aS-Module* to *anObject*. Now *anObject* also has a handle of *aS-Module* which the *S-ModuleSubject* does not know about.

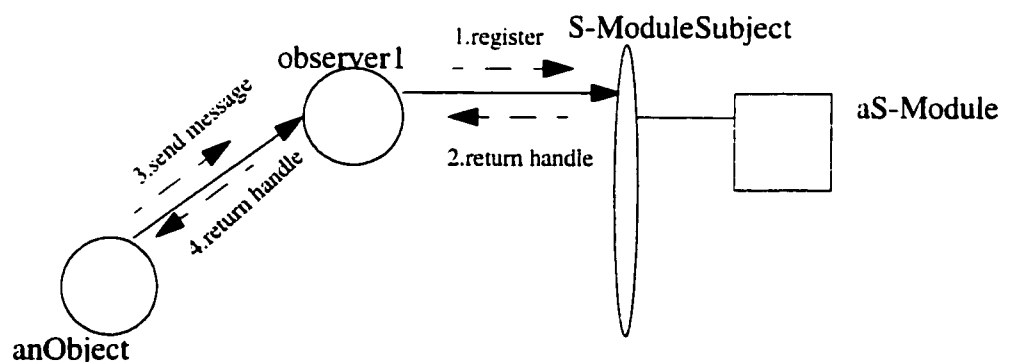


FIGURE 3-5. Object Reference Propagation Problem

One way to solve the problem of reference propagation or so called objects cascading is to propagate the design of the Observer Pattern too. *observer1* should also keep a record of to what objects it has returned the handle of *aS-Module*. Thus when *observer1* is called to update its reference to *aS-Module*, all the members in that record will also be asked to update accordingly.

Another way of solving this problem is for *observer1* to force *anObject* to register with the *S-ModuleSubject* automatically before *observer1* returns the handle of *aS-Module* to *anObject*.

Either way could work theoretically. However, they all put the burden of reference updating on the observers like *observer1*. *observer1* has the responsibility to propagate the updating of *aS-Module*. If it fails to do so the whole hot-swapping procedure fails. The situation will become worse when *observer1* has several references to update. Besides the potential performance problem this approach is also against the spirit of our hot-swapping technology, which is to upgrade objects or S-modules without affecting the other parts of the system. The swapping transaction should be controlled only within the Swap Manager and the S-modules, other parts of the system should not even notice the swapping activity, or at the very least should not participate in the swapping procedure.

3.3.2.2.2 Direct Reference Issue

In the Observer Pattern approach, objects have direct references to S-Modules, the subject (*S-ModuleSubject*) or the Swap Manager has no control over the access to S-Modules by other objects. In the worst case we could end up with different versions of the same kind

of S-Module co-existing at the same time, which may even cause the whole application to break down.

3.3.3 Pros and Cons

The Observer Pattern provides the abstract coupling between *subject* and *observer*. All a *subject* knows is that it has lists of *observers*, each conforming to the simple interface of the abstract Observer class. The *subject* does not know the concrete class of any *observer*. Thus the coupling between *subject* and *observers* is abstract and minimal.

The disadvantage of this approach is that it forces all the observers who want to access the services provided by an S-Module to register themselves to the subject of which the S-Module is a state, so that when the S-Module is hot-swapped all the observers could be notified and updated. This means that any object in the application program needs to follow certain design and implementation rules, if it wants to use the services of S-Modules. If it fails to do so, the whole hot-swapping transaction fails.

It is the programmers' responsibility to make the objects which want to use the services of S-Modules register with the S-Module's subject and keep track of the reference propagation. This will tremendously increase the complexities of the program and makes it very difficult to maintain.

Another disadvantage is the performance issue. Depending on how many observers are registered with the *S-ModuleSubject*, the notification procedure could take various time periods. Thus the performance of the hot-swapping system is application dependent.

3.4 The Proxy Pattern Approach

3.4.1 Proxy Pattern

The Proxy Pattern provides access control to an object by having a surrogate or placeholder for it [3].

Figure 3-6 shows a class diagram of the Proxy Pattern. The key objects in this pattern are *Proxy* and *subject*.

Proxy: maintains a reference that lets the *proxy* access the real *subject*; provides an interface identical to *subject's* so that a *proxy* can be substituted for the real *subject*; controls access to the real **subject**.

Subject: defines the common interface for *RealSubject* and *Proxy* so that a *proxy* can be used anywhere a *RealSubject* is expected.

RealSubject: defines the real object that the *proxy* represents.

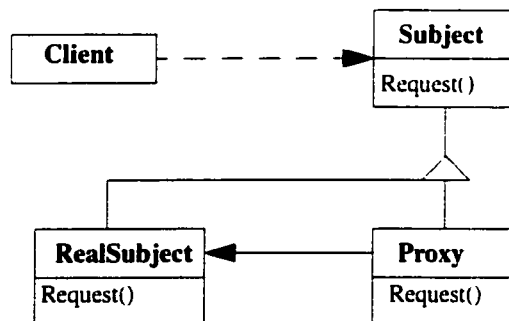


FIGURE 3-6. Proxy Pattern Structure

Figure 3-7 shows a possible object diagram at run-time. A protection proxy (*aProxy*) controls access to the original object (*aRealSubject*) by de-coupling the client (*aClient*) and a real subject (*aRealSubject*). *aClient* only has a reference to *aProxy*. only *aProxy* has a reference to *aRealSubject*.

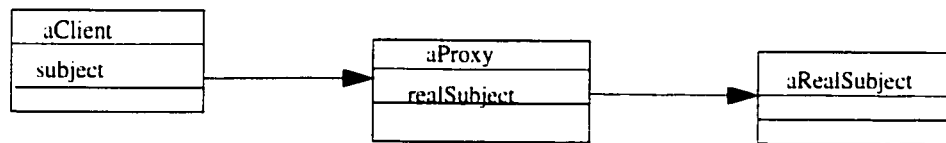


FIGURE 3-7. Proxy Pattern Object Diagram

3.4.2 Hot-swapping by Using the Proxy Pattern

3.4.2.1 Rationale

The main idea of having a proxy for each S-Module is to prevent direct access to the S-Module. This idea is similar to the JVM which acts as an abstract interface between the application and the underlying real computer.

Each S-Module has one proxy object (here we call it the S-Proxy object) associated with it. This is shown in Figure 3-8. An S-Module and its S-Proxy together comprise an S-Component. Thus the hot-swapping system contains one Swap Manager, several S-Components and other non S-Module objects. The S-Proxy is not swappable, it is created once when the program starts up and the corresponding S-Module object is instantiated.

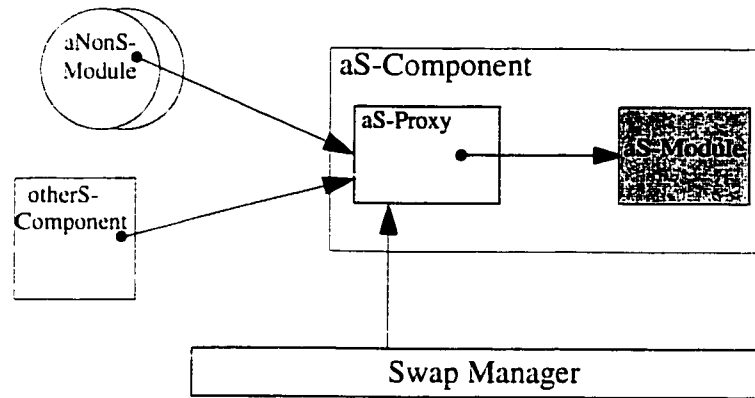


FIGURE 3-8. The Proxy Pattern Approach

A hot swapping scenario in the Proxy Pattern approach looks like this:

1. The Swap Manager gets a message from its listening service. The incoming message includes all the information about a new S-Module, including the identity, such as the object ID of an S-Module which is going to be hot-swapped, the version of the S-Module, and the code for instantiating objects, etc. (The protocol for messaging between the Swap Manager and the outside world will not be discussed in this thesis).
2. After the security service finishes the authentication checking with a successful result, the incoming new S-Module is instantiated to an S-Module object (here we call it a new S-Module. The object instantiation is the subject of other researches).
3. The Swap Manager searches for the S-Proxy of the current S-Module (here we call it the old S-Module) with the specific identity, and sends a message to the S-Proxy to block the new calls, and starts the timing service.

4. Then the Swap Manager checks with the S-Proxy if the old S-Module is in a swappable state. If no, which means it can not be swapped now, the Swap Manager will wait until all the interactions with this S-Module finish, or until a time-out event occurs.
5. When the old S-Module enters the swappable state, the Swap Manager calls the S-Proxy to get the internal states of the old S-Module, and maps them to the states of the new S-Module. (The mapping between different versions of S-Modules is discussed later).
6. After the new S-Module has been successfully initialized, the Swap Manager issues 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 if the transaction is finished within the required time limit. If yes, this hot-swapping is successful. The Swap Manager releases the blocked calls.
8. If during step 4 to step 6, the timing service gets a time-out event, this transaction is failed. The Swap Manager should send out a notification about this transaction and release the blocked calls.

3.4.2.2 Known Issue

The Proxy Pattern approach is based on the assumption that the proxy object (S-Proxy) is not swappable, while the real object (S-Module) it represents can be hot-swapped. The interfaces on the proxy objects are fixed. However, sometimes different versions of the S-Module may have different behavior methods, this will cause exceptions when the S-Proxy forwards the calls to the new S-Modules. Later in Chapter 5 , we will discuss how

to extend this approach to support the so called dynamic messaging function, which allows changes in some behavior methods of different versions of S-Modules.

3.4.3 Pros and Cons

The Proxy Pattern approach does not have the problem of reference propagating as discussed in the Observer Pattern approach, because all the client objects only have references (directly or indirectly) to the S-Proxy. Only the S-Proxy has a reference to the corresponding S-Module. The client objects and the S-Module are decoupled by the S-Proxy. The client objects should not be aware of the swapping transactions between the S-Proxy and the Swap Manager. This will tremendously reduce the complexity of client objects as contrast to the Observer Pattern approach.

Another advantage of the Proxy Pattern approach over the Observer Pattern approach is that it makes dynamic messaging possible. By using the Java reflection APIs, an S-Proxy can identify all the members that are associated with an S-Module and invoke the methods on it at run-time. This means that if the S-Proxy has an interface to accept dynamic messages, the client objects can use new services of the new S-Module which are not supported by the old S-Module at run time. However, the reflection process takes time and will affect the whole system performance.

In an S-Application, each S-Module object should have one S-Proxy object created for it. The hot-swapping transaction will not generate new S-Proxy objects. If the system has many S-Modules, the overhead of creating and managing the same number of S-Proxies

can not be ignored. This is again a trade-off between the flexibility and the performance of an application.

3.5 The Mediator Pattern Approach

3.5.1 Mediator Pattern

The Mediator Pattern defines an object that encapsulates how a set of objects interact with each other. A mediator promotes loose coupling by keeping objects from referring to each other explicitly, and it lets you vary their interaction independently [3].

The main two parts in the Mediator Pattern are one *mediator* and several *colleagues*. Figure 3-9 shows the structure of the Mediator Pattern.

Mediator: defines an interface for communicating with *Colleague* object.

ConcreteMediator: implements cooperative behavior by coordinating *Colleague* objects; knows and maintains all its *colleagues*.

Colleague: knows its Mediator object; communicates with its Mediator whenever it would have otherwise communicated with another colleague.

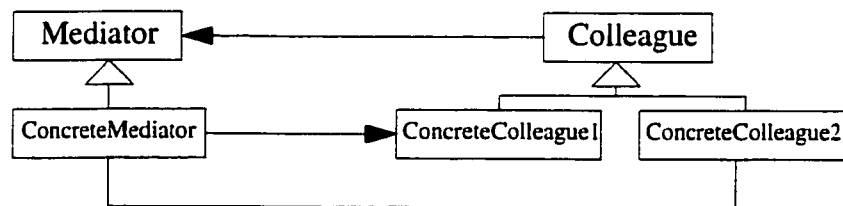


FIGURE 3-9. Mediator Pattern Structure

Figure 3-10 shows an object diagram of the Mediator Pattern. The *ConcreteMediator* has references to all the colleagues. *AColleague1* and *aColleague2* do not have references to each other. They send and receive requests from *aConcreteMediator*. The *aConcreteMediator* implements the cooperative behavior by routing requests between the appropriate colleagues.

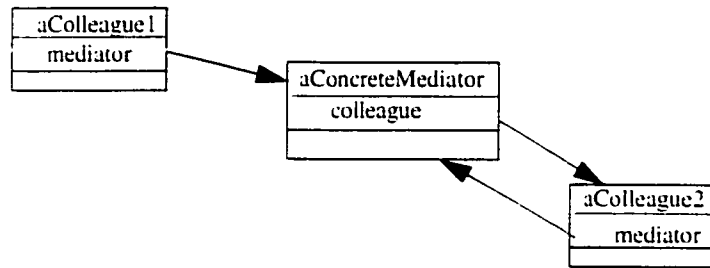


FIGURE 3-10. Mediator Pattern Object Diagram

3.5.2 Hot-swapping by Using the Mediator Pattern

3.5.2.1 Rationale

The main idea of the Mediator Pattern is to de-couple the colleague objects from referring to each other directly. The mediator works as a hub of communication (or so called active lookup service) which handles all the message interactions between the colleagues. This gives us a hint that we could use a mediator (*S-ModuleMediator*) in the hot-swapping system to handle the messaging between S-Modules and non-S-Modules. Since non-S-Modules and S-Modules are designed not to refer to each other directly, the hot-swapping transaction will take place only within the *S-ModuleMediator* and the Swap Manager. Each S-Module has a unique identity which could be used to identify itself among others.

As shown in Figure 3-11, a hot-swapping application using the Mediator Pattern approach includes a Swap Manager, several S-Modules, non-S-Modules, and a mediator called the *S-ModuleMediator*.

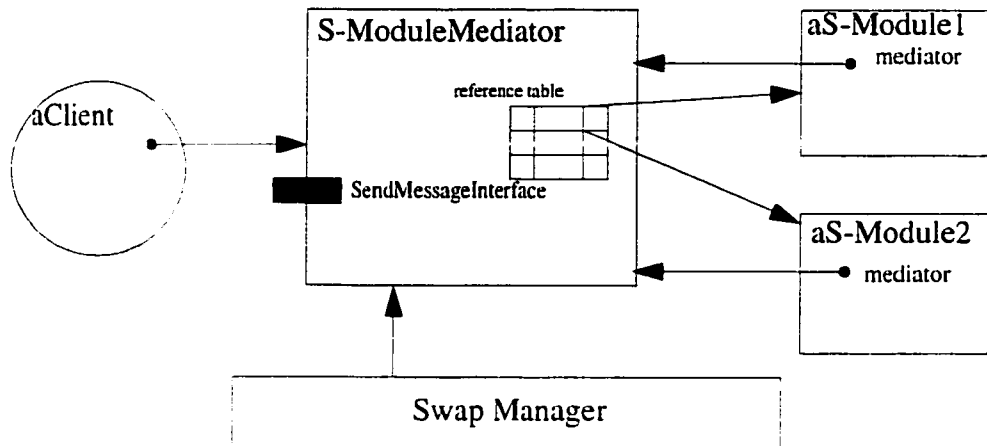


FIGURE 3-11. The Mediator Pattern Approach

The *S-ModuleMediator* has a table of references to all the S-Modules inside the application. The table is searched by the identity of each S-Module. The *S-ModuleMediator* is like a proxy for all the S-Modules. The difference between an *S-ModuleMediator* and an *S-Proxy* (as in the previous Proxy Pattern approach) is that all the references to all the S-Modules are centralized in one *S-ModuleMediator*, not in each *S-Proxy*.

The *S-ModuleMediator* has a service interface (*SendMessageInterface*) for all clients (S-Modules and non-S-Modules) when they want to interact with S-Modules. Each time when a client wants to send a message to an S-Module, it has to call the *S-ModuleMediator*'s *sendMessage* method and pass-in the target S-Module's identity and the required service (the method name and its parameters). Then the *S-ModuleMediator* will lookup the

corresponding S-Module in its reference table and send the message to it. This lookup procedure will use the Java reflection [7] in order to achieve the goal of dynamic messaging.

A hot-swapping scenario in the Mediator Pattern approach looks like this:

1. The *Swap Manager* gets a message from its listening service. The incoming message includes all the information about a new S-Module, including the identity, such as the object ID of an S-Module which is going to be hot-swapped, the version of the S-Module, and the code for instantiating objects, etc.
2. After the security service finishes the authentication checking with a successful result, the incoming new S-Module (the byte code) is instantiated to an S-Module object (here we call it a new S-Module).
3. The *Swap Manager* searches for the corresponding S-Module (here we call it the old S-Module) with the specified identity in the *S-ModuleMediator*, and sends a message to the *S-ModuleMediator* to block the new calls to this S-Module, and starts the timing service.
4. Then the *Swap Manager* checks with the *S-ModuleMediator* if the old S-Module is in a swappable state. If no, which means it can not be swapped now, the Swap Manager will wait until all the interactions with this S-Module finish.
5. When the old S-Module enters the swappable state, the *Swap Manager* calls the *S-ModuleMediator* to get the internal state of the old S-Module, and maps it to the state in the new S-Module.

6. After the new S-Module has been successfully initialized, the *Swap Manager* issues a hot-swapping transaction by changing the reference to the old S-Module in the *S-ModuleMediator* with the reference to the new S-Module.
7. The timing service stops timing and checks if the transaction is finished within the required time limit. If yes, this hot-swapping is successful. The *Swap Manager* releases the blocked calls.
8. If during the step 4 to step 6, the timing service gets a time-out event, this transaction is failed. The *Swap Manager* should send out a notification about this transaction and release the blocked calls.

3.5.2.2 Known Issues

The Mediator Pattern approach takes advantage of *java.lang.Class* and the reflection API. The mediator (here the *S-ModuleMediator*) has references to all the S-Modules' *Class* (*java.lang.Class*) instances. Once you have a reference to a *Class* instance, you have access to a wealth of run-time information about that object's class in the JVM. On the other side, since the *S-ModuleMediator* has such a super power in an application, it must be designed and implemented as robust and secure as possible.

3.5.3 Pros and Cons

The Mediator Pattern approach uses Java reflection heavily therefore it will suffer from performance problem.

The Mediator Pattern trades complexity of interaction for the complexity of a centralized control object. This can make the mediator itself a monolith that's very hard to maintain.

The advantage is that it reduces the coupling between S-Modules. A S-ModuleMediator replaces many-to-many interactions with one-to-many interaction between the S-Module-Mediator and all the client objects and S-Modules. Hot-swapping an S-module is transparent to other S-Modules and objects.

Also, the Mediator Pattern approach makes dynamic messaging easy to handle.

3.6 Conclusions

In this chapter four approaches for designing the S-Module in an S-Application are discussed.

The Java Virtual Machine approach goes deep into the implementation of a virtual machine and sacrifices the platform independency which we definitely want to avoid.

The Observer Pattern approach is useful in notifying all the registered observers. But the subject does not de-couple the observers with the S-Modules. Also the potential indirect reference is very hard to control. However, the mechanism of notification could be adopted by other approaches.

The Proxy Pattern approach seems to be more reasonable than other approaches. It also could have the functionality of dynamic messaging. However, it is based on the assumption that the S-Proxy is not swappable.

The Mediator Pattern approach centralizes the control of all the S-Modules which in turn makes the mediator complex to maintain. The mediator has a simple interface for clients to call but use the Java reflection heavily. The disadvantage of this approach is the inefficiency caused by using Java reflection, thus is not suitable for performance critical applications.

Nothing comes for free. The trade-off between flexibility and efficiency is application dependent. As in a small application like the Modular SNMP project, the Proxy Pattern approach could be a better choice. By comparison these potential approaches, the Proxy Pattern approach appears to be more promising than the others. Because of its simple structure and easy maintenance, the Proxy Pattern approach is chosen as the best candidate to be deployed at the current stage of research. However, as the research of hot-swapping goes on, other new approaches or mixed approaches may show up. In this thesis, the Proxy Pattern approach is selected to be developed in detail, and is described in the next chapter.

Chapter 4 Designing S-Modules by Using Proxy Pattern

This chapter provides a framework for designing of Java programs with S-Modules. It specifies the general rules for designing and implementing S-Modules and the corresponding S-Proxies by using the Proxy Pattern approach. Figure 4-1 defines the research target and environments.

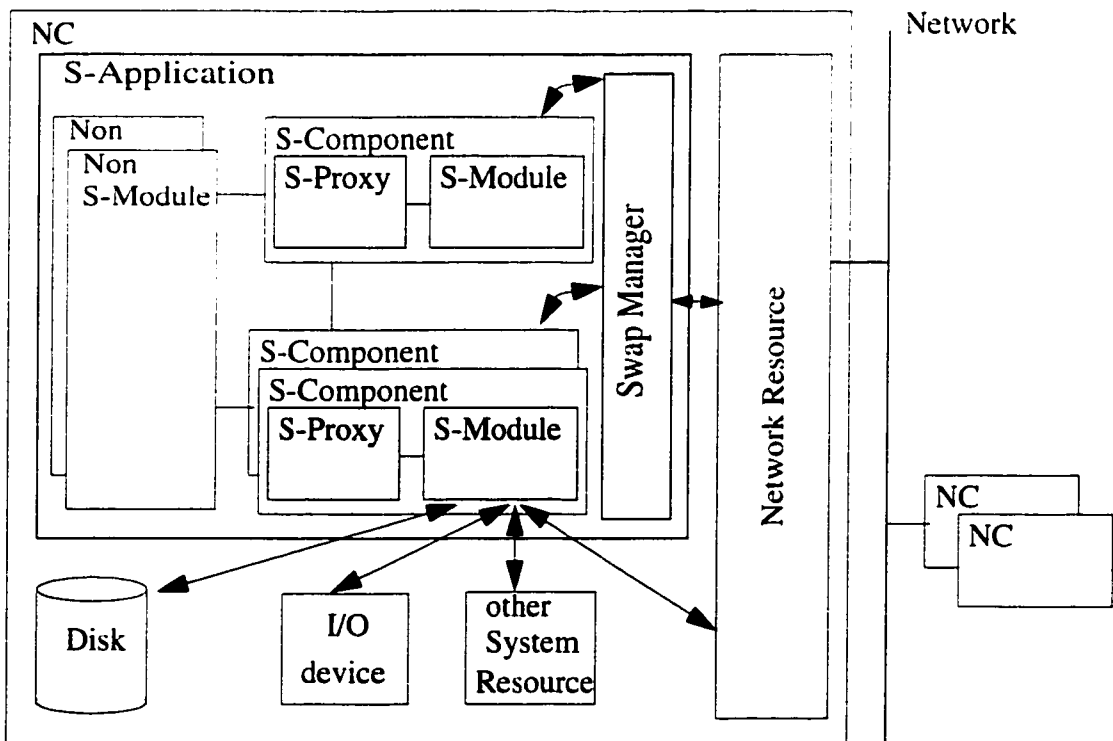


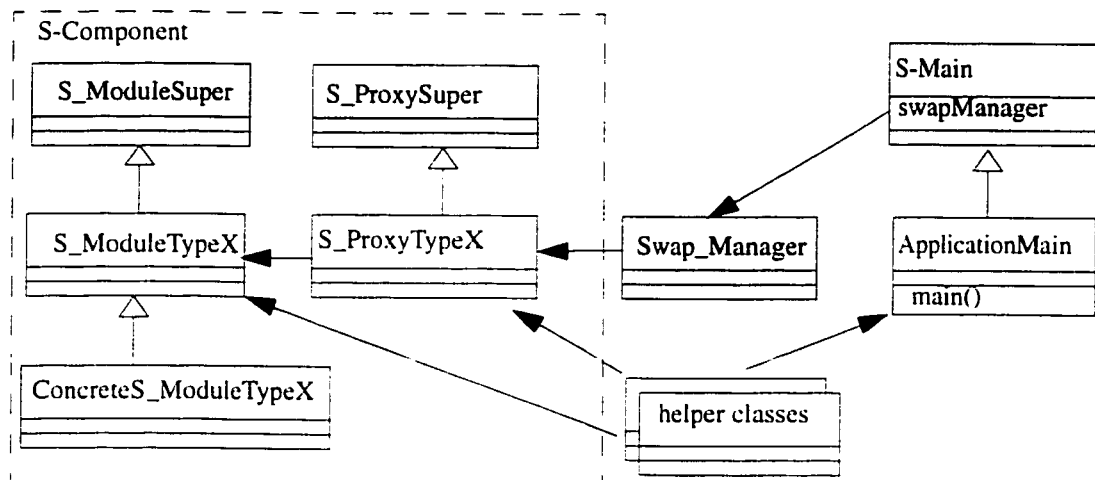
FIGURE 4-1. Designing S-Module by Using Proxy Pattern

As shown in Figure 4-1, each S-Application program is composed of S-Components (an S-Component is made up of an S-Module and its S-Proxy) and non-S-Modules, it also has a Swap Manager which monitors and controls all S-Components. S-Components interact with each others as well as with non-S-Modules. In this chapter, we concentrate on the research of designing an S-Module and its S-Proxy. The Swap Manager is also an important component of the program, but we will not address all aspects of the Swap Manager in this thesis. The basic functions of Swap Manager are described, but not in a detailed fashion. S-Modules could also interact with its environments through JVM (not drawn in the figure), we identify the following resources which an S-Module may access:

- Disk and files,
- Other I/O devices, such as serial ports,
- System resources, such as timers, and
- Network resources, such as sockets.

4.1 Logical View

Java is a real OO language, an application program written in Java is made up of classes and class instances. A logical view depicts the relationship between classes of a program. Figure 4-2 is a general logical view for any program which has S-Components (S-Module and S-Proxy) inside.



Note: Shaded classes are framework classes and will be defined in this chapter.

FIGURE 4-2. Logical View of Programs with S-Modules

An S-Application program should have one and only one class instance which has the *main()* method implemented. The class, called *ApplicationMain* class (in fact, the designer can choose any name he/she wants), should inherit from a super class called *S_Main* which will initialize the program to accommodate S-Components. The *S_main* class is provided by the hot-swapping framework.

The *S_Main* class instantiates one and only one instance of the *Swap_Manager* class and calls the *start()* method on it. The *Swap_Manager* class is derived from the *java.lang.Thread*¹ class. When the *start()* method is called on the instance of the *Swap_Manager*, a Swap Manager thread will start running separately from the main thread. The *Swap_Manager* will keep running until the termination of the program. It is

1. *java.lang.Thread* is from Java core API of Sun Javasoft.

the Swap Manager that handles all the hot-swapping transactions. The *Swap_Manager* class and its implementation is provided by the framework.

In an S-Application program there can exist multiple instances of an S-Component (which includes an S-Module and an S-Proxy). The design of a concrete S-Module, as shown in Figure 4-2, follows a three level class hierarchy: an *S_ModuleSuper* class, an *S_ModuleTypeX* class, and a *ConcreteS_ModuleTypeX* class.

- The *S_ModuleSuper* class is an abstract class which defines all the attributes and common interfaces for all S-Modules. It is the highest level class in the S-Module class hierarchy. It is the root for any S-Module in the scope of the Proxy pattern approach. A definition of the *S_ModuleSuper* class is given later.
- The *S_ModuleTypeX* class is an abstract class for all S-Modules of the same type (type X) such as the SNMP security S-Modules. It inherits from the *S_ModuleSuper* class and defines the common behavior interface for the same type (type X) of S-Modules to implement. Rules for designing such a class are given later in this chapter.
- The *ConcreteS_ModuleTypeX* class is a concrete class and is the lowest level in the S-Module class hierarchy. It inherits from the *S_ModuleTypeX* class and implements the behavior interface of that type of S-Module.

The *S_ModuleSuper* class is provided by the framework, but it is the software designer's duty to design the *S_ModuleTypeX* class and implement the *ConcreteS_ModuleTypeX* by complying the rules described in this chapter. A mechanism should be invented to publish the *S_ModuleTypeX* interface, so that it is available to public. The research on this issue is far beyond the scope of this thesis, and will not be addressed here.

In contrast to the S-Module, the S-Proxy classes have only two levels: an *S_ProxySuper* class and an *S_ProxyTypeX* class. The *S_ProxySuper* class is an abstract class which defines the common attributes and interfaces for all the S-Proxies. For the same type of S-Modules, such as *S_ModuleTypeX*, there will be one corresponding S-Proxy such as the *S_ProxyTypeX* class. The *S_ProxyTypeX* class is derived from the *S_ProxySuper* class. It provides the same behavior interface as the corresponding *S_ModuleTypeX* does. Any access to the S-Module is done through the corresponding S-Proxy, only the S-Proxy has a handle to the S-Module object.

The *S_ProxySuper* class is provided by the framework. It is the software designer's duty to design and implement the *S_ProxyTypeX* class. Generally, The *S_ModuleTypeX* class and *S_ProxyTypeX* class should be provided in pair.

Other helper classes in the application program are normal java classes constructing the program. They will not be affected by the S-Module designing rules, and will not be discussed in detail here.

4.2 Rules for Designing An S-Module

As discussed in Section 3.1 , an S-Module should have certain characteristics, such as identity, service, internal state, dependency list, mapping rule, persistence and administration interface, etc. In this section, we will specify the rules for defining and implementing those characteristics.

4.2.1 Identity

An S-Module has a unique existence within a certain scope of the software society. The definition of the identity of an S-Module employs a layered structure. An S-Module can be uniquely identified by the combination of the following three attributes:

- Swapping model type,
- S-Module type, and
- S-Module descriptor.

A class definition for the S-Module identity looks like this:

```
public class SModuleIdentifier {
    private String swapping_model_type = "Proxy";
    private String s_module_type = "";
    // S-Module descriptor
    private SModuleDescriptor descriptor;
    ...//accessor methods
}
```

As a general rule, S-Modules are swappable only if they have the same Swapping model type and the S-Module type.

4.2.1.1 Swapping Model Type

As we discussed in previous chapter, there are different approaches towards the goal of software hot-swapping, such as the Observer Pattern approach, the Proxy Pattern approach, the Mediator Pattern approach, etc. Different approaches define different hot-swapping models. A *String* type variable will be used to represent those different models.

For the Proxy Pattern model, the one we specify in this chapter, the swapping model type is *Proxy*.

4.2.1.2 S-Module Type

A *String* type variable will be used to represent the type of S-Module to which a concrete S-Module belongs. For example, the S-Module type of a security module in an SNMP v3 application will be *SNMPv3 Security Module*.

4.2.1.3 S-Module Descriptor

An S-Module descriptor is used to identify a concrete S-Module of a certain type. It is made of:

- Name of this concrete S-Module, a *String* type variable such as *User Based Security Model Module*,
- Version number of this concrete S-Module, a *String* type variable such as *v1.1.0*,
- Name of the vendor, a *String* type such as *SCE of Carleton Univ*, and
- Date of release, a *String* type such as *1999-05-01*.

A class definition for the S-Module descriptor looks like this:

```
public class SModuleDescriptor{
    private String name; // "User-based Security Model
                        Module";
    private String version; // "v1.1.0";
    private String vendor; // "SCE of Carleton Univ";
    private String date; // "1999-05-01";
    ... //accessor methods
}
```

4.2.2 Service

An S-Module provides a set of services which show the behavior or functionality of the S-Module. There exist two types of service provided by an S-Module represented as *BehaviorInterface* and *SwappingServiceInterface*. The *BehaviorInterface* specifies the functions which can be accessed by other modules (through the corresponding S-Proxy) in the program. The *SwappingServiceInterface* is a framework defined interface, which allows others (the Swap Manager and the corresponding S-Proxy) to retrieve and set internal states/attributes of this S-Module. The *SwappingServiceInterface* is also defined as the administration interface in Section 3.1 . It is mandatory to be implemented by any S-Module.

4.2.2.1 BehaviorInterface

An S-Module has some kind of behavior when interacting with other S-Modules and its environment. S-Modules of a certain type should have the same common behavior (called published behavior), although different concrete S-Modules may also provide other specific behaviors (called vendor specific behavior).

The published behavior of an S-Module is specified in an abstract interface called the *BehaviorInterface*. S-Modules of the same type, such as the *S_ModuleTypeX*, must implement the same interface of that type, such as the *BehaviorInterfaceTypeX*, although the implementations in different concrete S-Modules are usually different.

An example of the *BehaviorInterfaceTypeX* class is listed as follows:


```
// example: published behavior for a security S-Module
public interface BehaviorInterfaceTypeX {
    public Object encryption ();
    public Object decryption ();
    ...
}
```

The vendor specific behavior of an S-Module is implemented by each concrete S-Module. Since different versions of the same type S-Module may provide different vendor specific behavior, which are not defined in the common *BehaviorInterface*, an S-Module must have the ability to deal with the exceptional situation when a call is made to a method that does not exist in this version but may be supported by another version. This is done by implementing an interface called *DoesNotUnderstandInterface*.

```
public interface DoesNotUnderstandInterface{
    public Object doesNotUnderstand (
        String methodName,
        Object[] args);
}
```

The *DoesNotUnderstandInterface* defines a method called *doesNotUnderstand()* which accepts the name of the vendor specific method and the necessary parameters as the method arguments. The implementation of this method could be very simple such as do nothing, or could be very complex. We only provide the interface definition, but will not give any guideline for how to handle an unknown method call. The implementation will be highly vendor dependent.

This interface is mandatory to be implemented by any S-Modules.

4.2.2.2 SwappingServiceInterface

The *SwappingServiceInterface* is an abstract interface that defines the methods which are used by the Swap Manager or the corresponding S-Proxy during a hot-swapping transaction. The following methods have been defined in the *SwappingServiceInterface*:

- The *getIdentifier()* method returns the identity information of the S-Module.
- The *getState()* method returns the current state of the S-Module.
- The *getAttributes()* method (with no argument) returns all the information about the attributes of the S-Module.
- The *getAttributes()* method (with the S-Module version in string type as argument) returns the information about the attributes in the required format as specified in the passed-in version. (There is a limitation to this method, which will be discussed in Section 4.2.5)
- The *mappingAttributes()* method takes an S-Module attributes as argument and maps the passed-in attributes with the ones defined in this S-Module. A detailed discussion about mapping attributes of different S-Modules is described later.

A definition of the *SwappingServiceInterface* is as follows:

```
public interface SwappingServiceInterface{
    public SModuleIdentifier getIdentifier();
    public String getState();
    public SModuleAttribute getAttributes();
    public SModuleAttribute getAttributes(
        String sModuleVersion);
    public void mappingAttributes(
        SModuleAttribute attributes);
}
```

4.2.3 Internal State

An S-Module has a finite number of internal states. None of these internal states will last indefinitely. Currently, five internal states are identified for an S-Module: *Initializing*, *Swappable*, *Swapping*, *Busy* and *Blocked*. An S-Module will use the access methods defined in *S-ModuleSuper* class to get and set its internal states accordingly (see Section 4.4.1).The hot-swapping transaction will take place if and only if an S-Module is in its *swappable* state.

4.2.3.1 Initializing State

In this state, an S-Module is being loaded into the program, and its internal data structures being initialized. After the initialization, the S-Module will set itself to the *Swappable* state by using the *setSModuleState()* method.

4.2.3.2 Swappable State

An S-Module enters the *swappable* state when it is idle, which means an S-Module is neither being called nor is it calling others, or actively using any system resources currently. Although it could be holding some system resources such as files, I/O devices, sockets etc., the S-Module is not using them to perform any operations at the time. Only in this state can a hot-swapping transaction take place immediately. After being swapped out the S-Module will be garbage collected later.

4.2.3.3 Busy State

In this state, one or more methods of an S-Module are being called by other modules in the program, but the S-Module itself is not requesting any service from other modules or system resources.

4.2.3.4 Blocked State

In this state, the S-Module itself is calling other modules of the program, or using the system resources, and waiting for the completion of the operations. If an S-Module just simply holds system resources such as opened files, but is not using it such as reading/writing data from/into file, it is not in the *Blocked* state, it is in the *Busy* state.

4.2.3.5 Swapping State

When an S-Module is in either the *Busy* state or the *Blocked* state, and receives a request to be swapped, the S-Module is forced to enter the *Swapping* state. In the *Swapping* state the S-Module should either continue its normal operations or be asked to terminate its services in a safe way and release the system resources it holds. After that, the S-Module enters the *Swappable* state.

4.2.3.6 State Diagram

Figure 4-3 depicts the state transitions of an S-Module. When an S-Module is first loaded or swapped into the program, it will enter the *Initializing* state. After initialization, the S-Module enters the *Swappable* state which is in fact in idle and waits for service requests

(method calls). When one or more services of the S-Module are requested (methods are called), it enters the *Busy* state until the requested services are completed and goes back to the *Swappable* state again. In the *Busy* state, it may enter the *Blocked* state in which the S-Module is waiting for other modules or system to complete its service requests. As a general rule, any S-Module should not stay in the *Busy* and/or the *Blocked* state in an infinite time, which will block the whole hot-swapping transaction. Many programming technologies can be used to avoid a certain software module being in the *Busy* or the *Blocked* state forever, for example, instead of using the blocking model TCP socket, we can design the system to use the non-blocking model of TCP socket (a call back function is needed).

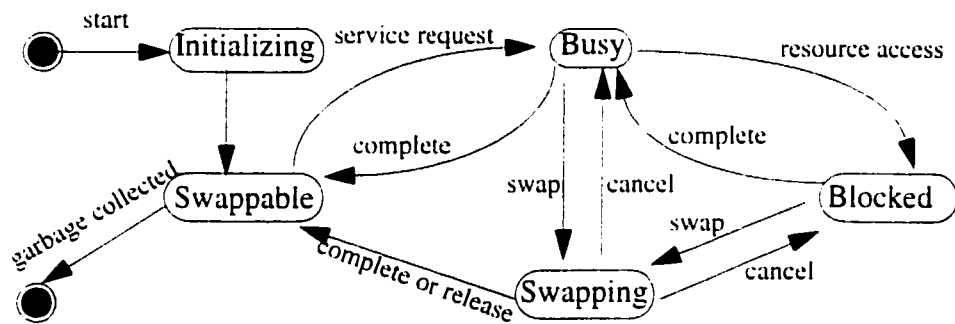


FIGURE 4-3. State Diagram

An S-Module can enter the *Swapping* state from the *Busy* state or the *Blocked* state. When entering the *Swapping* state from the *Busy* or the *Blocked* state, the S-Module can continue the current services normally or terminate them in a safe way. The S-Module can go back to the *Busy* or the *Blocked* state from the *Swapping* state upon the cancellation of a swapping transaction.

Upon the success of a swapping transaction, the swapped-out S-Module will be garbage collected from the *Swappable* state, and the swapped-in S-Module will enter its *Swappable* state (through *Initializing* state).

4.2.4 Dependency List

An S-Module has a dependency list which records all the S-Modules that this S-Module depends on. Before an S-Module can be activated, all the S-Modules that it depends on must have been loaded into the program (in a certain order, this will be handled by the Swap Manager).

```
public class DependencyList {
    Vector dependencyList;//a list of dependents
    Vector getDependencyList();
    void addDependent(SModuleIdentifier id);
};
```

Each S-Module which is being depended on is identified by the S-Module identity. A wildcard can be used to specify a group of S-Modules. For example, "SNMPv3 Security" S-Module *A* depends on S-Module *B* (type = "Authentication", version = "v1.1.*"), this means S-Module *B* from version v1.1.0 to version v1.1.9 can be used.

When the Swap Manager loads S-Modules, it needs to derive the loading order and to check the deadlock condition by drawing a dependency map for involved S-Modules. We use an arrow line to represent a dependent relationship. If S-Module *A* depends on S-Module *B*, an arrow line is drawn from *A* to *B*. At any time, if a circle can be found in the map, the deadlock happens. Starting from one S-Module, goes along one arrow line to another S-Module (according to the direction of arrow). Repeating this procedure for each S-Mod-

ule on the road, if reaching the starting S-Module again, a circle is found and that's a deadlock condition. Figure 4-4 gives an example of a dependency map and shows a deadlock condition.

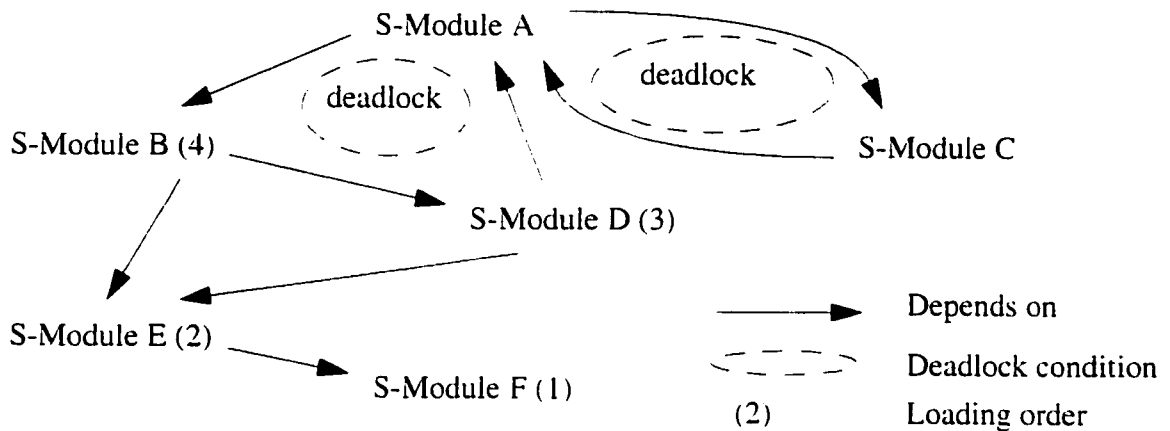


FIGURE 4-4. Dependency Map and Deadlock Condition

As a general rule, if and only if no deadlock condition in a dependency map exists, the involved S-Modules can be swapped into the program.

It is also the Swap Manager's duty to find out the loading order of S-Modules through the dependency map. As in the above figure, if we only take S-Module *B*, *D*, *E* and *F* into consideration, then the correct loading order should be: *F*, *E*, *D* and *B*.

The research and analysis of the algorithm used to calculate deadlock condition and loading order is beyond the scope of this thesis, and will not be addressed here.

4.2.5 Mapping Rule and Persistency

One of the challenging issues in the hot-swapping transaction is to translate the attributes from one S-Module to another S-Module. The attributes are in fact the class or instance variables of a certain concrete S-Module. In order to maintain the persistency during the hot-swapping transaction, sometimes it is necessary to map the attribute values from the “old” S-Module to the “new” S-Module. The attributes of two S-Modules may have different quantities and different types. Not all the attributes are needed to be mapped during a hot-swapping transaction. The selection of attributes being mapped is S-Module dependent and also vendor specific. However, a general guideline for the mapping rules is given below:

- For attributes of the Java primitive types, such as *int*, *byte*, *short*, *long*, *float*, *double*, *char*, and *boolean* types, the mapping rules are straight forward: 1) the value of the attribute could be assigned directly to the primitive type attribute of another S-Module, or 2) by applying some calculations, map the attribute to the primitive type attribute of another S-Module.

```
// for example:  
int S_Module1_counter;  
int S_Module2_counter;  
// mapping  
S_Module2_counter = S_Module1_counter;
```

- For attributes of the class type, including the class types defined in the Sun’s Java core class library and other helper classes, the attribute is a reference to an instance of a certain type of classes. During a hot-swapping transaction, this attribute could be assigned

to another S-Module's attribute which is of the same class type or of its super class type.

```
// for example:  
String S_Module1_str;  
String S_Module2_str;  
// mapping  
S_Module2_str = S_Module1_str;
```

- For attributes of a certain class types, such as those holding system and network resources, since the S-Module is in a swappable state the references to these resources could be copied to the new S-Module's attributes of the same type or the resource could be released without affecting the service of the S-Module.

```
// for example:  
Socket S_Module1_socket1;  
Socket S_Module1_socket2;  
Socket S_Module2_socket;  
// mapping  
S_Module2_socket = S_Module1_socket1;  
S_Module1_socket2.close();
```

There are three cases of attributes mapping:

- a mapping from an S-Module to the same type of S-Module with a newer version from the same vendor.
- a mapping from the current S-Module to the same type of S-Module with an older version from the same vendor
- a mapping between S-Modules from different vendors.

There are be different approaches to handle the attributes mapping. In this section we discuss two approaches: one is that an S-Module handles the mapping by implementing the mapping rules inside, the other approach is that the Swap Manager handles the mapping.

4.2.5.1 Mapping Rules Inside the S-Module

In this approach the mapping rules have been hard coded inside each S-Module. When the Swap Manager calls certain methods on the S-Module, the attributes are mapped automatically. The mapping rule is transparent to the Swap Manager.

A method in the *SwappingServiceInterface* called *mappingAttributes()* must be implemented to solve the mapping from an S-Module to a newer version. When the Swap Manager calls this method on a newer version S-Module, it passes in the attributes of the old version S-Module. It is the newer version S-Module's responsibility to recognize and map these attributes to its new attributes according to the mapping rules.

A method called *getAttributes()* (with the previous version of S-Module as an argument) can be implemented when there is a need to swap backwards to any previous version of the S-Module. The Swap Manager will call this method to retrieve the attributes in the form of any requested previous version, which is passed in as an argument. An S-Module must define and implement all the mapping rules for mapping to each previous versions.

There is a restriction to this approach, an S-Module must know the design of its previous versions so that it can implement mapping rules associated with them. This is only possible when different versions of S-Modules are from the same vendor.

4.2.5.2 Mapping Rules Outside the S-Module

Another approach is to free the S-Module from implementing all the mapping rules especially in the case of mapping backwards, or to map between different vendor's S-Modules. Since different vendors can provide different versions of S-Modules, it is impossible to have S-Modules of different vendors to implement the mapping rules for each other. In this case, the Swap Manager will retrieve the attributes of the current running S-Module, apply a certain mapping rule in order to convert them to a required format and assign them to the new attributes of the new S-Module. The Swap Manager should be able to set the mapping rules manually, for example through an user interface, so that it can handle all mapping cases.

One drawback of this approach could be a complicated design of the Swap Manager, also it will take longer time for the processing of a hot-swapping transaction at run-time. Another limitation of this approach is that it can hardly be used in the embedded system, where providing a user interface is almost impossible.

4.3 Rules for Designing an S-Proxy

An S-Proxy provides interfaces for the clients of this S-Component as well as interfaces for the Swap Manager. We define clients as other S-Modules or non-S-Modules which want to use the services of an S-Module. Only the S-Proxy has a reference to the corresponding S-Module. The clients can only contact the S-Proxy when they require the services of the corresponding S-Module. In other words, as long as the S-Proxy remains static, the existence of the corresponding S-Module is transparent to its clients.

An S-Proxy should have the following features:

- Attributes
- Interfaces for clients
- Interfaces for the Swap Manager

4.3.1 Attributes

An S-Proxy has several attributes which indicate the status of the corresponding S-Module and could be used by the Swap Manager to control the access to the S-Module.

- An object reference to the corresponding S-Module: An S-Proxy has one and only one reference to the current S-Module. During a hot-swapping transaction, this reference will be changed to refer to the new S-Module object. The value (object handle) of this reference should not be passed out of the S-Proxy to any clients at any time.
- A *boolean* type condition variable named *not_in_swap*: If *not_in_swap* is *TRUE*, which means that the S-Module is not marked to be swapped, then all the clients (in a multi-threading application it refers to all the clients in all running threads) of this S-Component could call the methods in the *BehaviorInterface* and the *NewBehaviorInterface*, it is the S-Module's responsibility to handle the concurrence issue; if *not_in_swap* is *FALSE*, which means that the Swap Manager has marked the S-Module to be swapped some time later, then all the calling threads coming afterwards will be blocked at the S-Proxy until the hot-swapping transaction is finished and the Swap Manager sets the condition variable to be *TRUE* again. The value of this condition variable could only be changed by the Swap Manager.

4.3.2 Interfaces for Clients

4.3.2.1 BehaviorInterface

An S-Proxy must implement the same published *BehaviorInterface* as the corresponding S-Module, such as the *BehaviorInterfaceTypeX*. The implementation of the methods in the *BehaviorInterfaceTypeX* first checks the condition variable *not_in_swap*. If *not_in_swap* is *TRUE* then it forwards the calls from the clients to the corresponding S-Module; if *FALSE*, the newly incoming calls will be blocked until the condition variable becomes *TRUE*.

4.3.2.2 NewBehaviorInterface

As mentioned in Section 4.2.2.1, an S-Module can define and implement vendor specific service interfaces which are not documented by the common *BehaviorInterface*, thus is not published and supported by the S-Proxy. In order to invoke these un-published methods of an S-Module, a specific interface will be defined in the S-Proxy so that the S-Proxy can invoke dynamic methods on the corresponding S-Module at run-time. This interface is called *NewBehaviorInterface*.

The *NewBehaviorInterface* allows client objects to send dynamic messages (make method calls) at run-time to the methods which do not exist at the compile time in the *BehaviorInterface* of the corresponding S-Component (S-Module and S-Proxy).

Java, unlike other languages such as Smalltalk, does not permit a method to be invoked that doesn't actually exist, with the resulting error trapped in a *doesNotUnderstand*

method. This so called dynamic messaging is not possible in Java, as all methods are type-checked by the compiler [6].

However, with the power of Java reflection APIs [7], [8], [9], we can add a dynamic messaging functionality by implementing the *NewBehaviorInterface* in the S-Proxy.

The *java.lang.reflect* package gives us the ability to identify all of the members that are associated with an object (field members and method members) and makes it possible for that object to interact with them. For each S-Module, all the methods and the fields could be checked and retrieved at run-time. If a required method is not found an exception will be caught by the S-Proxy and the exception is handled by the *DoesNotUnderstandInterface* of the corresponding S-Module.

The *NewBehaviorInterface* is an interface for the external client objects to send dynamic messages to the S-Proxy. The client objects can call the *newBehaviorMethod()* and the call will be passed to the new S-Module's requested method with the required method name and the parameters.

The implementation of the *NewBehaviorInterface* will first check the condition variable. If *TRUE*, it then checks whether the required method (with the passed-in method name and the arguments) is supported by the corresponding S-Module class. If yes the call is passed to that method of the S-Module. If not supported by this S-Module, the call is sent to the *DoesNotUnderstandInterface* of the S-Module. If the condition variable is *FALSE*, this call to the *NewBehaviorInterface* is blocked temporally.

A definition of the *NewBehaviorInterface* looks like this:

```
public interface NewBehaviorInterface{
    public Object newBehaviorMethod(String
                                   methodName, Object[] args);
}
```

4.3.3 Interfaces for the Swap Manager

The Swap Manager will contact each S-Proxy through the following interfaces. The implementation of these interfaces in each S-Proxy is mandatory.

4.3.3.1 SignalInterface

The *SignalInterface* is used by the Swap Manager to mark and un-mark an S-Module to be swapped. The *setCondition()* method sets the condition variable *not-in-swap* to be *TRUE* or *FALSE*, while the *getCondition()* method returns the value of *not-in-swap*.

A definition of the *SignalInterface* looks like this:

```
public interface SignalInterface {
    public void setCondition(Boolean b);
    public Boolean getCondition();
}
```

4.3.3.2 GetAttributesInterface

The *GetAttributesInterface* is used by the Swap Manager to retrieve the attributes of the corresponding S-Module during a hot-swapping transaction. The implementation of this interface in the S-Proxy forwards the call from the Swap Manager to the corresponding S-Module's *getAttributes()* method and return the necessary attributes to the Swap Manager.

A definition of the *GetAttributesInterface* looks like this:

```
public interface GetAttributesInterface {
    public SModuleIdentity getIdentifier();
    public SModuleAttribute getAttributes();
    public SModuleAttribute getAttributes(String
                                        sModuleVersion);
}
```

4.3.3.3 SwapControlInterface

The *SwapControlInterface* should only be called by the Swap Manager. Through this interface the Swap Manager controls the transaction of the hot-swapping procedure from start to end. Security should be applied to ensure that only the Swap Manager can use this interface.

The *prepareSwap()* method is used to mark the S-Module and block new calls when the Swap Manager starts the hot-swapping transaction. It returns *TRUE* when the S-Module enters the *swappable* state. The *swap()* method changes the reference to the old S-Module in the S-Proxy with the reference to the passed-in new S-Module. The *releaseCall()* method un-marks the S-Module and releases the blocked calls.

A definition of the *SwapControlInterface* looks like this:

```
public interface SwapControlInterface {
    public boolean prepareSwap();
    public void swap(S-Module newS-Module);
    public void releaseCall();
}
```


4.4 Major Abstract Classes Definition

According to the discussion above, in this section we give the definition for three major abstract classes in the hot-swapping framework, the S_ModuleSuper class, the S_ProxySuper class, and the S_Main class. Also a guideline for the S_ModuleTypeX class and S_ProxyTypeX is given.

4.4.1 S_ModuleSuper Class

The S_ModuleSuper class defines the interfaces that all the S-Modules must implement.

An example of the definition of this class looks like this:

```
public abstract class S_ModuleSuper implements
    Swappable,
    SwappingServiceInterface,
    DoesNotUnderstandInterface {
    static SModuleIdentifier sModuleID;
    private DependencyList dList;
    private String state = "Initializing";
    public SModuleIdentifier getSModuleID(){
        return sModuleID;
    }
    public void setSModuleState(String s){
        state = s ;
    }
    public DependencyList getDependencyList(){
        return dList;
    }
    public void setDependencyList(DependencyList l){
        dList = l;
    }
}
```

4.4.1.1 Swappable

Swappable is an interface with no methods, thus no implementation is needed. The usage of this interface in the *S_ModuleSuper* class is as a marker on any S-Modules to indicate that an S-Module is a swappable component.

The Swap Manager can check if a software component is an S-Module by testing whether it implements the *SwappableInterface*.

4.4.1.2 SwappingServiceInterface

SwappingServiceInterface is an abstract interface described in Section 4.2.2.2 .

4.4.1.3 DoesNotUnderstandInterface

DoesNotUnderstandInterface is an abstract interface described in Section 4.2.2.1 .

4.4.1.4 Attributes

Several attributes and their accessor methods are defined in the *S_ModuleSuper* class:

- A *SModuleIdentifier* type variable called *sModuleID*, which keeps all the identification information about an S-Module.
- A *DependencyList* type variable called *dList*, which stores the dependency list of this S-Module.
- A *String* type variable called *state*, which indicates the current state of the S-Module such as “Initializing”, “Busy”, “Blocked”, “Swappable” and “Swapping”.

4.4.2 S_ProxySuper Class

The S_ProxySuper class defines all the attributes and interfaces that all the S-Proxy classes must implement. An example of the definition of this class looks like this:

```
public abstract class S_ProxySuper implements
    NewBehaviorInterface,
    SignalInterface,
    GetAttributesInterface,
    SwapControlInterface {
private S_ModuleSuper sm;
private Condition not_in_swap;
}
```

An explanation of the interfaces which the S_ProxySuper class implements is in Section 4.3.2 and Section 4.3.3 .The definitions of the attributes are described in Section 4.3.1 accordingly.

4.4.3 S_Main Class

The S_Main class is a super class from which an application class that has a *main()* method inside should sub-class. The only thing defined in this class is the creation of an instance of the Swap_Manager class. When an application class which derives from the S_Main class starts up, a Swap Manager thread is generated and starts running. The Swap Manager thread will stop execution when the main thread ends.

An example of this class looks like this:

```
public class S_Main {
    private Swap_Manager swapManager;
    public S_Main(){
        swapManager = new Swap_Manager();
        swapManager.start();
    }
}
```

4.4.4 S_ModuleTypeX

The S_ModuleTypeX class should inherits the S_ModuleSuper class and implements a corresponding BehaviorInterfaceTypeX. A template of the S_ModuleTypeX looks like this:

```
public abstract class S_ModuleTypeX
    extends S_ModuleSuper
    implements BehaviorInterfaceTypeX {
}
```

4.4.5 S-ProxyTypeX

The S_ProxyTypeX class should inherits the S_ProxySuper class and implements the same BehaviorInterfaceTypeX as the corresponding S_ModuleTypeX class. A template of the S_ProxyTypeX looks like this:

```
public class S_ProxyTypeX extends S_ProxySuper
    implements BehaviorInterfaceTypeX {
    //implementation goes here
}
```

4.5 Component Diagram

In summary, the relationship between the Swap Manager, an S-Component (includes an S-Module and its S-Proxy) is shown in Figure 4-5.

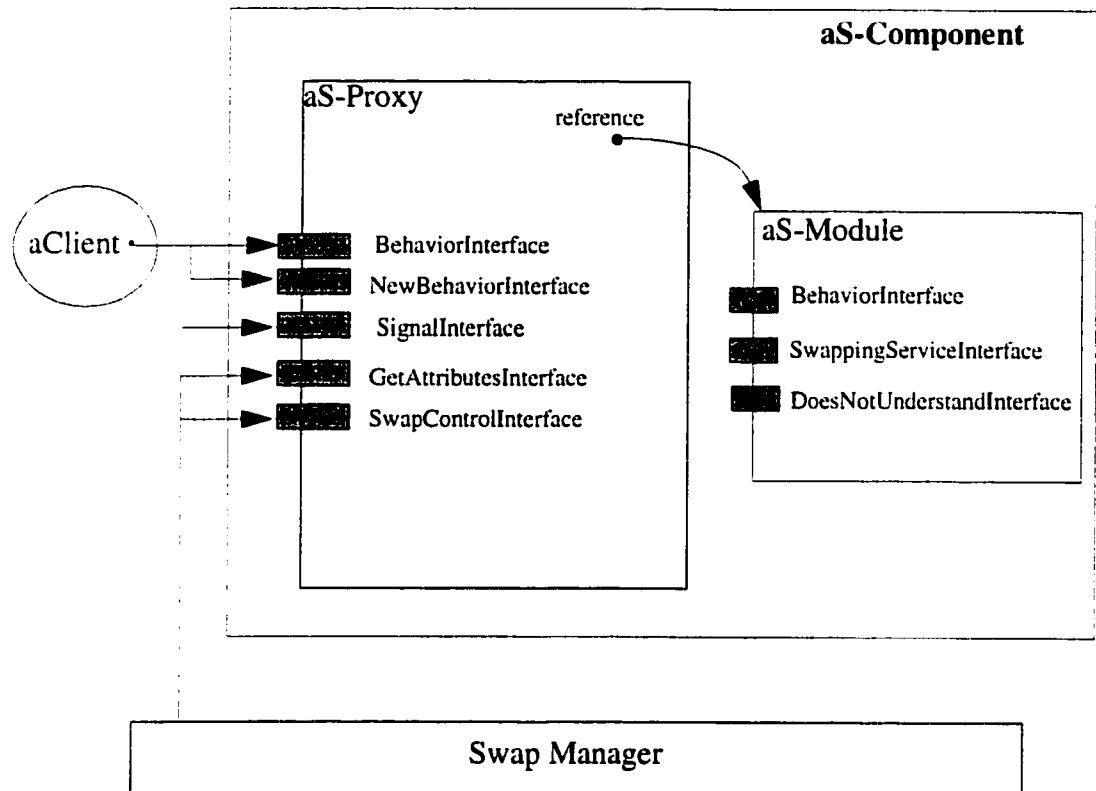


FIGURE 4-5. Interfaces In an S-Module And Its S-Proxy

4.6 Sequence Diagram

A sequence diagram which focuses on the hot-swapping transaction between the involved S-Modules, the corresponding S-Proxy, the Swap Manager, and a client object is shown in Figure 4-6.

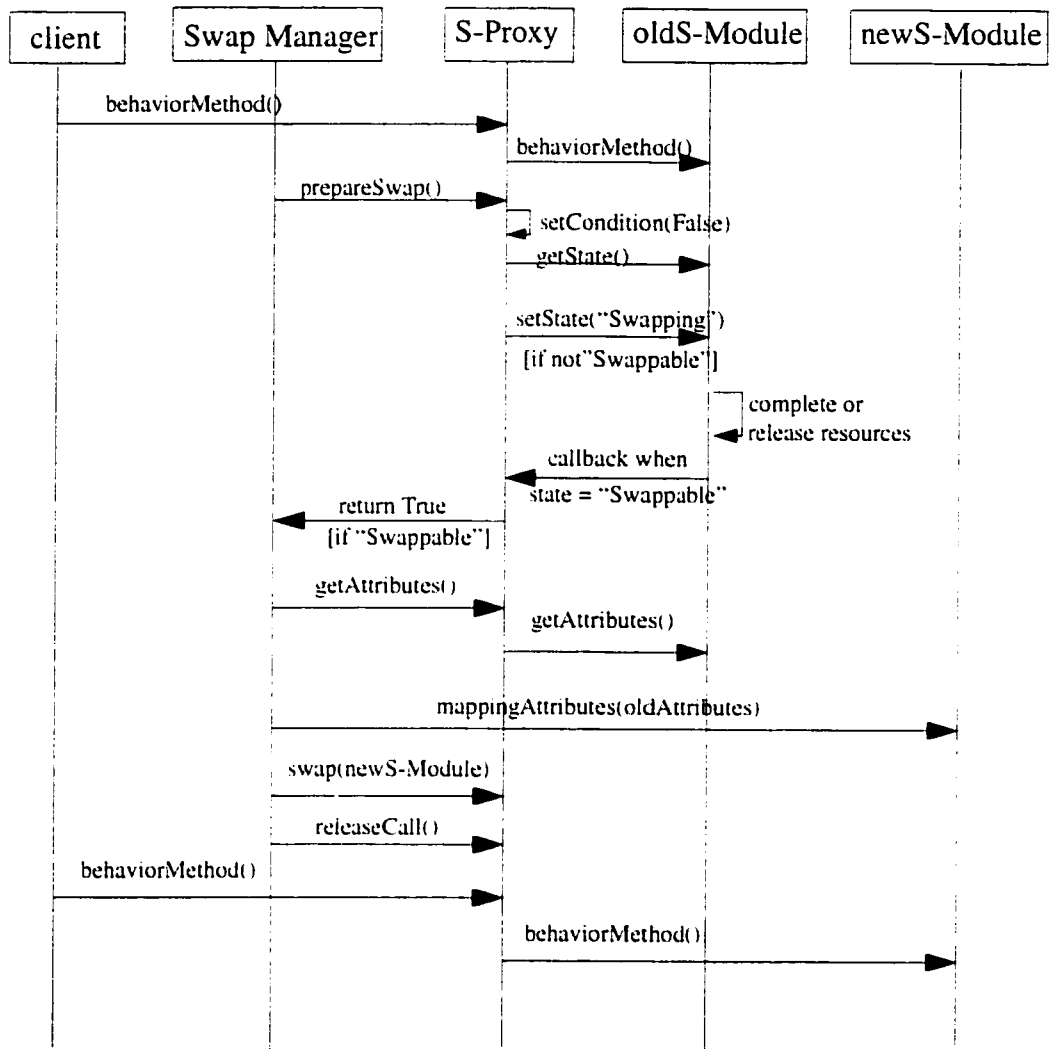


FIGURE 4-6. Sequence Diagram

4.7 Scenarios

Once an S-Module has been hot-swapped, clients begin to interface with it. The following sections describe three distinct scenarios that are possible after a hot-swapping transaction takes place.

It is worth mention again that Java can only do syntax checking, not semantics checking. Therefore, if method A from the old S-Module and method B from the new S-Module have the same syntax (both method name and parameters) but provide different service, the Java Virtual Machine can not distinguish them.

4.7.1 New and Old S-Modules Have Same Behavior Methods

This is a simple scenario and should happen in most cases. The two versions of the same type of S-Module support not only the exactly same *BehaviorInterface* but also the same vendor specific behavior methods which can be invoked through the S-Proxy's *NewBehaviorInterface*. After the hot-swapping, calls from clients are forwarded by the S-Proxy to the new S-Module's corresponding methods. The activity inside the S-Proxy will be the same as with both the old and the new S-Module.

As shown in Figure 4-7, a client (*aClient*) sends a message to the S-Proxy (*aS-Proxy*) by calling the *SameMethod* in the *BehaviorInterface* with parameter *args*. Then *aS-Proxy* forwards the call to the just swapped-in S-Module (*newS-Module*) directly.

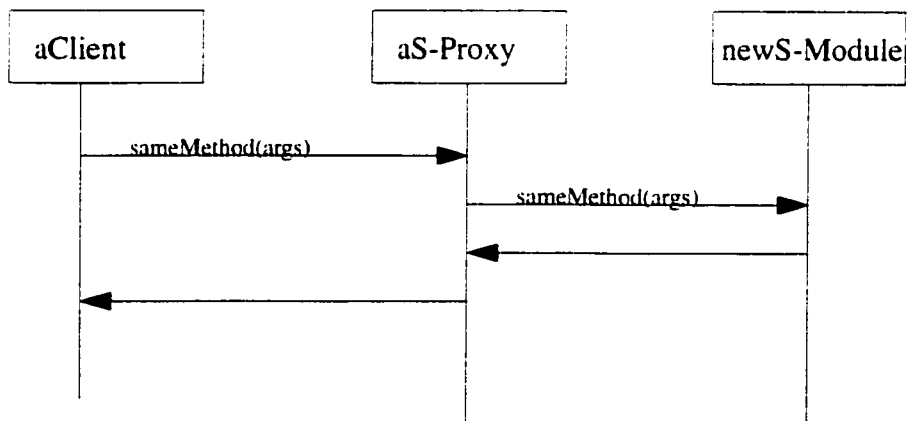
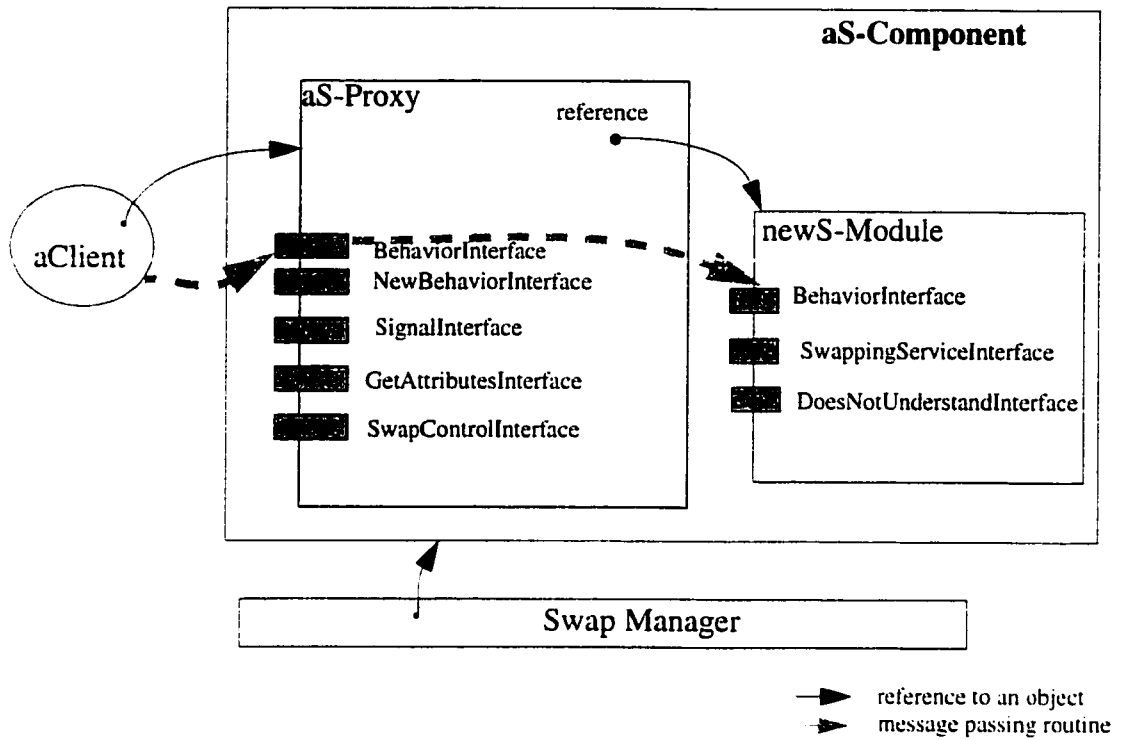


FIGURE 4-7. New and Old S-Module Have Same Behavior Methods

4.7.2 New S-Module Does Not Have All Old S-Module Methods

As we discussed in Section 4.2.2.1 , an S-Module could have some vendor specific behavior methods that are not included in the published *BehaviorInterface*. Therefore it is possible that the new S-Module does not support some of the vendor specific methods of the old S-Module, or simply because the new S-Module requires different parameters for a particular method. However, the clients may not be aware of the difference between the old and the new S-Module, so they may still call the vendor specific methods through the *NewBehaviorInterface*.

Figure 4-8 shows this scenario. Here, a vendor specific behavior method of the old S-Module is the *oldMethod*. *aClient* sends a message to the *aS-Proxy* through the *NewBehaviorInterface* giving the name of the required method (*oldMethod*) and *arg* as the parameters. The *aS-Proxy* checks with the *newS-Module* and finds out that the *newS-Module* does not support this *oldMethod*. Then the *aS-Proxy* sends a message to the *newS-Module*'s *DoesNotUnderstandInterface* with the method name (*oldMethod*) and the parameter (*arg*). From now on it is the *newS-Module*'s responsibility to diagnose and make further decisions.

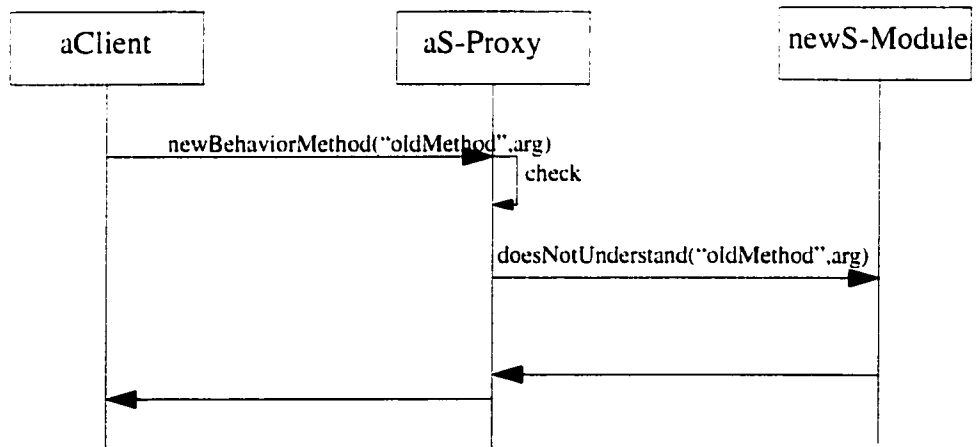
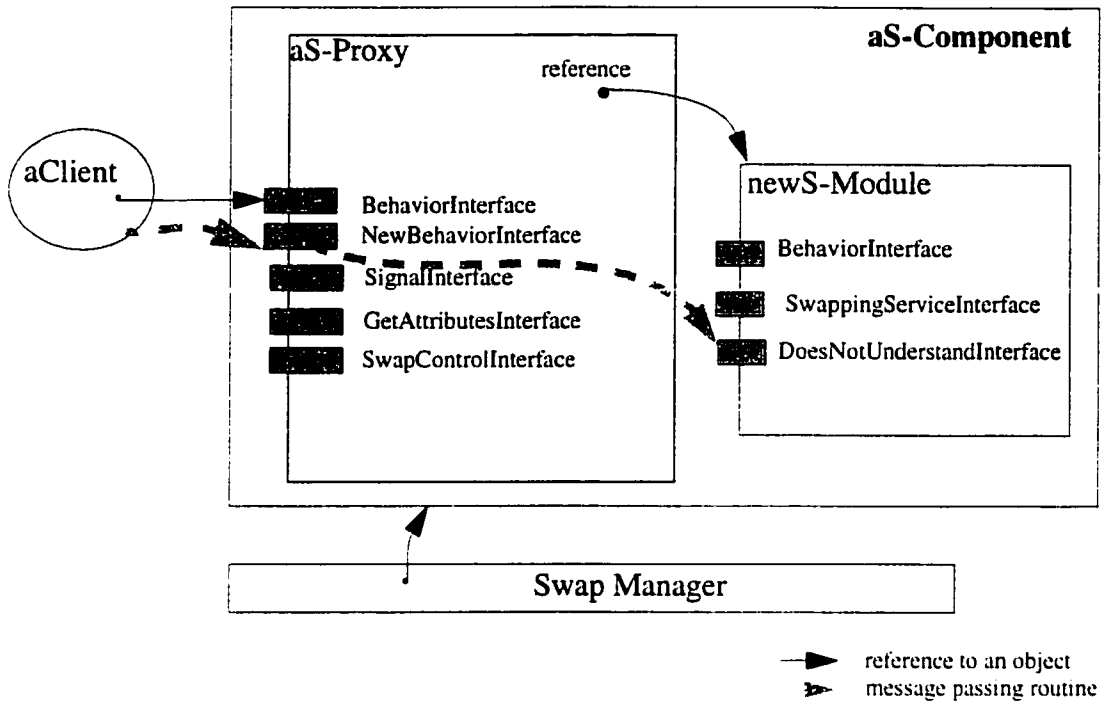


FIGURE 4-8. New S-Module Does Not Have All Old S-Module Methods

4.7.3 New S-Module has methods that old S-Module does not

In this case, the new S-Module has added some new vendor specific methods. This scenario is more likely to happen when the caller is also an S-Module that has been hot-swapped.

As shown in Figure 4-9, the *newS-Module* has a new vendor specific behavior method called *newMethod* which requires *arg* as the parameter. A client S-Module (here we call it *clientS-Module*) sends a message to *aS-Proxy*'s *NewBehaviorInterface* with the name *newMethod* and the *arg*. The *aS-Proxy* checks with the *newS-Module*. If the required *newMethod* is supported by the *newS-Module*, the *aS-Proxy* invokes the *newMethod* on the *newS-Module* with the parameter *arg*. If not, *aS-Proxy* forwards the call with the method (*newMethod*) and the parameters (*arg*) to the *newS-Module*'s *DoesNotUnderstandInterface*. It is again the *newS-Module*'s responsibility to diagnose and make further decisions.

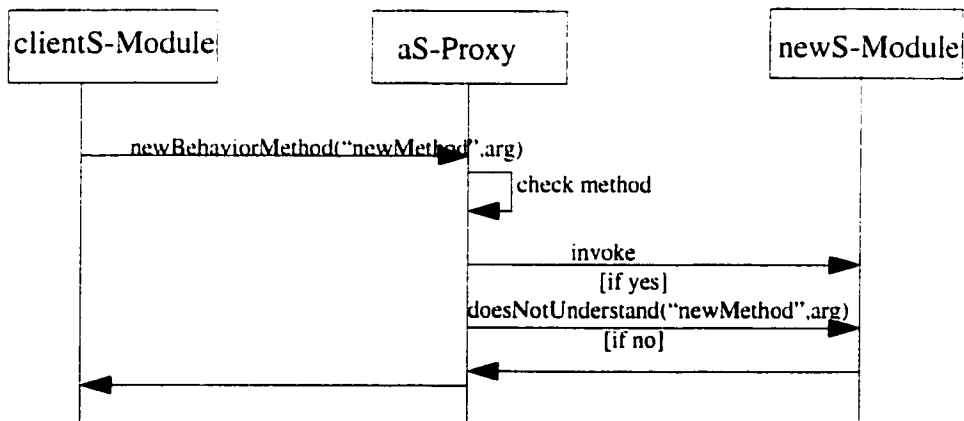
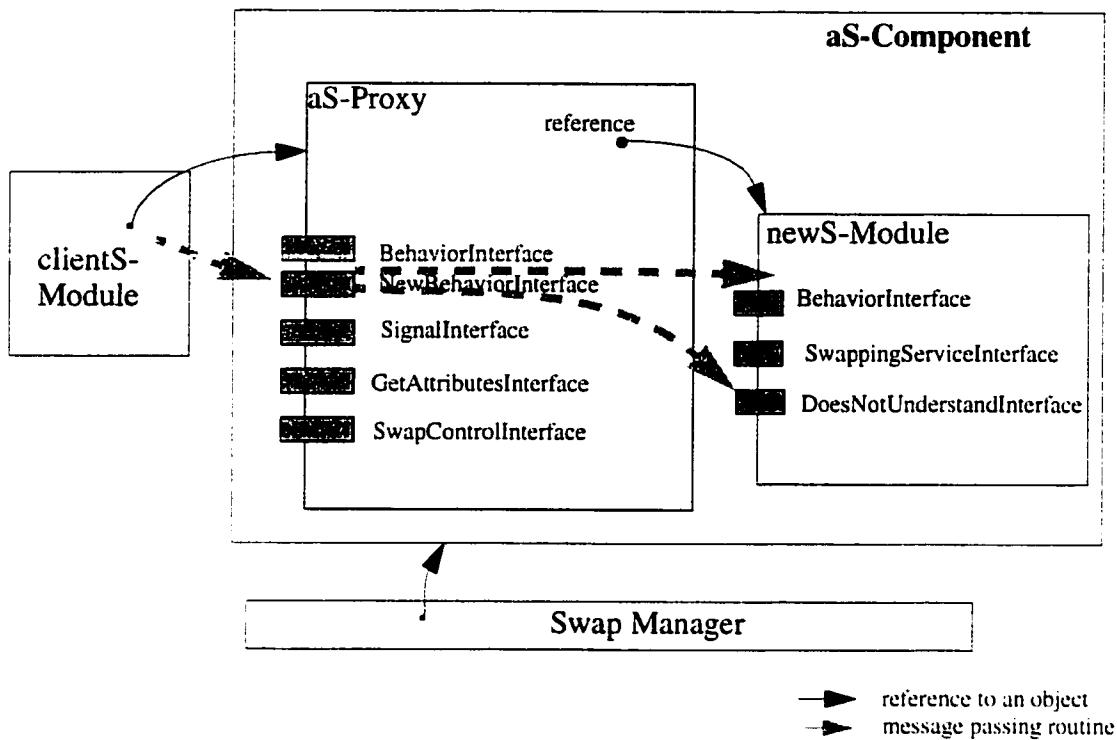


FIGURE 4-9. New S-Module Has Methods That Old S-Module Does Not

Chapter 5 Hot-swapping Application: A Swappable SNMP Agent

This chapter demonstrates how to apply the hot-swapping framework as discussed in the previous chapter to a real world application. An SNMPv3 agent in the traditional SNMP-based communication network management system has been decomposed and re-constructed into an S-Module based S-Application. The advantages of modifying SNMPv3 network management system into a hot-swapping system are as follows:

- Although the SNMPv3 has already been implemented by several vendors and research groups [10], it is still an immature standards track protocol and there remains many outstanding minor questions of interpretation that are being ironed out and will continue to be resolved for some time [11]. The requirements for dynamic upgrading or bug fixing one or several subsystems in an SNMPv3 engine will remain in recent years. Modifying such a system not only provides a test-bed for the hot-swapping framework, but also has practical value in the real world.
- On the other hand, the SNMPv3 is a well modularized system - made up of several subsystems with pre-defined interfaces which describe the interactions between the modules. This modularized system architecture fits well into our hot-swapping framework.

Since this thesis concentrates on the research of software hot-swapping technology, not on SNMP itself, we did not implement everything from scratch. Instead, the application of the SNMP network management system is based upon an original Java implementation of SNMPv3 from UQAM [12]. This implementation model is a modular approach [13] and follows the interfaces as defined by the RFCs [15], [16], [17], [18], [19].

In the following sections a brief overview of the SNMP architecture is first introduced, then the design and implementation of an SNMP agent consisting of S-Modules is described. Finally, the test cases on the swappable SNMP agent application and the test results are discussed.

5.1 SNMP Architecture

5.1.1 Background Information

Since its first publication in 1988, the Simple Network Management Protocol (SNMP) has become the most widely-used network management protocol on TCP/IP-based communication networks. SNMP defines a protocol for the exchange of management information, a format for representing management information and a framework for organizing distributing systems into managing systems and managed agents. In addition, a number of specific data structures, called Management Information Bases (MIBs), have been defined as part of the SNMP suite; these MIBs specify managed objects for the most common network management subjects, including bridges, routers, and Local Area Networks (LANs) [11].

SNMPv1 became both an open Internet Engineering Task Force (IETF) standard and a de facto industry resulting from widespread market acceptance. A board 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 most obvious weakness of SNMPv1 is the lack of adequate mechanisms for security, such as the lack of authentication and privacy mechanisms as well as an administrative framework for authorization and access control. The IETF working group that developed SNMPv2 wanted to include security functionality in the new version but was not able to reach agreement on how to define the required security mechanism.

The third version of SNMP (SNMPv3) is derived from and builds upon both the first and the second versions. The new IETF SNMPv3 working group has produced a set of Proposed Internet standards published as RFCs 2571-2575 [15], [16], [17], [18], [19]. This document set defines a framework for incorporating security features into an overall capability that includes either SNMPv1 or SNMPv2 functionality as well. Also, the documents define a specific set of capabilities for network security and access control. Meanwhile, it is for the first time a modular SNMP engine is defined.

5.1.2 SNMP Architecture

All versions of SNMP (v1, v2, and v3) share the same basic structure and components, and follow the same architecture. The SNMP framework emphasizes the use of modular-

ity for the evolution of portions of SNMP without requiring a redesign of the general framework.

The SNMP architecture, as described in RFC2271[15], consists of a distributed, interacting collection of SNMP entities. Each entity implements a portion of the SNMP capability and may act as an agent, a manager, or a combination of the two. Each SNMP entity includes a single SNMP engine and some applications. An SNMP engine implements functions for sending and receiving messages, authenticating and encrypting/decrypting messages, and controlling access to managed objects. These functions are provided as services to one or more applications that are configured with the SNMP engine to form an SNMP entity. An SNMP engine contains a Dispatcher, a Message Processing Subsystem, a Security Subsystem and an Access Control Subsystem.

- The Dispatcher is a simple traffic manager. It sends and receives messages to/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 Protocol Data Unit (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. An implementation of the Message Processing Subsystem may support a single message format corresponding to a single version of SNMP (SNMPv1, SNMPv2, SNMPv3), or it may contain a number of modules, each supporting a different version of SNMP [16].

- The Security Subsystem provides services such as the authentication and privacy of messages. An implementation of the Security Subsystem may contain multiple Security Models (currently defined User-based Security model (USM) [18] and future other Security models). A security model defines the threats against which it protects, the service provided and the security protocols used (procedure and MIB data) to provide the service such as authentication and privacy.
- The Access Control Subsystem provides authorization services to control access to MIBs for the reading and setting of management objects. These services are performed on the basis of the contents of PDUs. An implementation of the Access Control Subsystem may support one or more distinct access control models. So far the only defined security model is the View-based Access Control Model (VACM) for SNMPv3 [19].

5.1.3 User-Based Security Model

The User-Based Security Model for SNMPv3 [18] defines the elements of procedure for providing SNMP message-level security. USM protects against following primary and secondary threats: modification of information, masquerade, message stream modification, and (optionally) disclosure. Two cryptographic functions are defined for USM: authentication and encryption. USM allows the use of one of the two alternative authentication protocols: HMAC-MD5-96 and HMAC-SHA-96. USM uses the cipher block chaining (CBC) mode of the Data Encryption Standard (DES) for encryption.

5.1.4 Abstract Service Interface

The services between modules in an SNMP entity are described as abstract service interfaces defined in the RFCs in terms of primitives and parameters. A primitive specifies the function to be performed, and the parameters are used to pass data and control information. As an example, Figure 5-1 shows the sequence of events in which an agent responds to an incoming request. The figure shows how an incoming message results in the dispatch of the enclosed PDU to an application, and how that application's response results in an outgoing message. Also the figure demonstrates how the primitives fit together. Note that some of the arrows in the diagram are labeled with a primitive name, representing a call. Unlabeled arrows represents the return from a call.

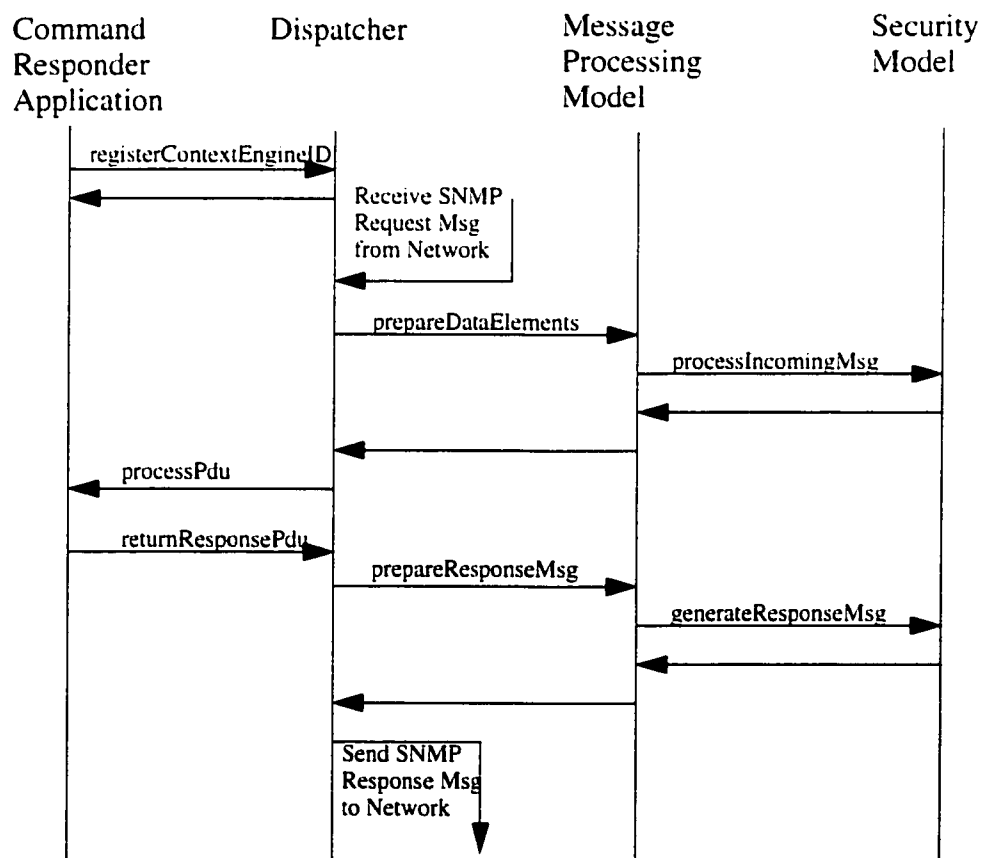


FIGURE 5-1. Command Responder

5.2 Swappable SNMP Agent

5.2.1 System Architecture

As mentioned in Section 5.1.2 , a traditional SNMP agent is composed of several SNMP applications, such as Command Responder Application, Notification Originator Application, and Proxy Forwarder Application, and an SNMP engine. An SNMP agent also has some MIB implementations to support the management applications.

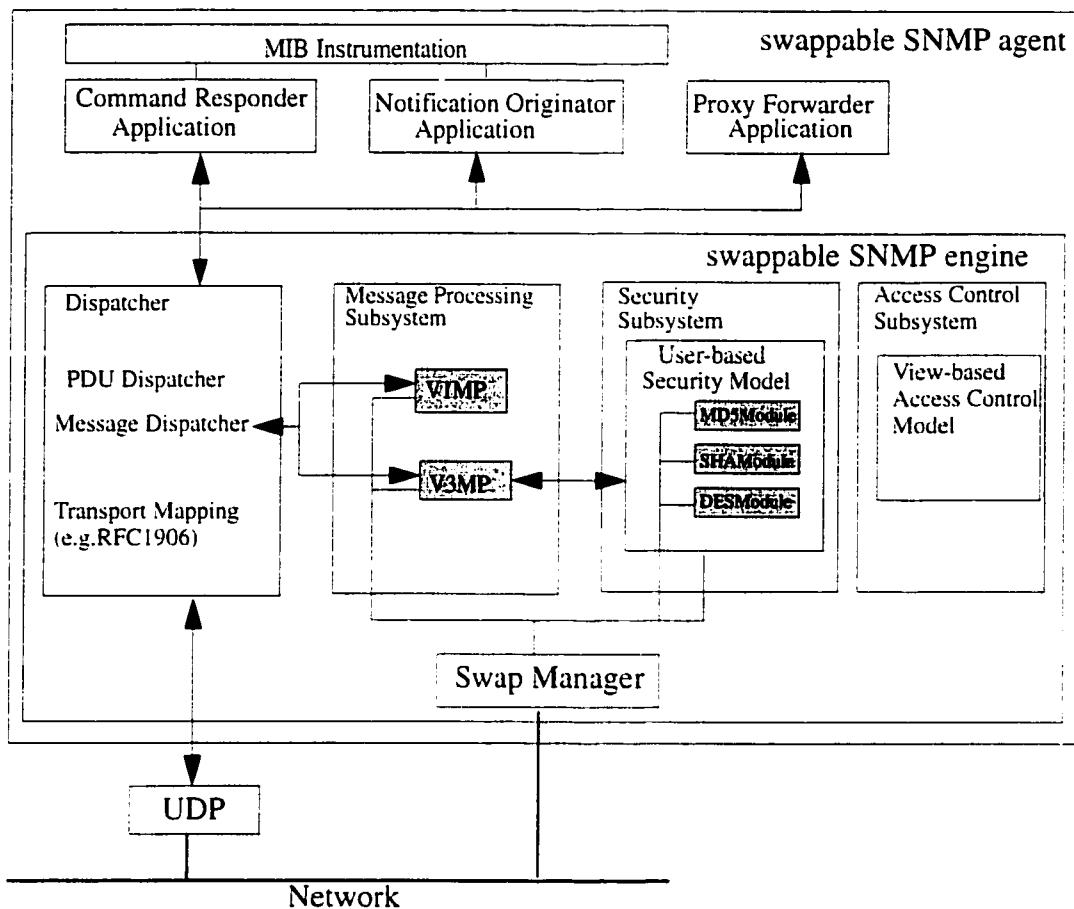
When an SNMP agent receives an incoming message, the Dispatcher accepts the message from the transport layer and routes it to the appropriate message processing module. The appropriate message processing module processes each message header, and returns the enclosed PDU to the Dispatcher if it is a v1 message, or, passes it to the appropriate security model if it is a v3 message. In case of a v3 message, if required, the security model checks the authentication code, performs decryption, and returns the processed message to the message processing module, which then returns the enclosed PDU to the Dispatcher. The Dispatcher then passes this PDU to the appropriate application.

For an outgoing response, the Dispatcher accepts a PDU from the application, determines the type of message processing required (i.e., SNMPv1, SNMPv3) and passes the PDU on to the appropriate message processing module in the Message Processing Subsystem. The appropriate message processing module prepares the PDU for transmission by wrapping it in the appropriate message header and returning it to the Dispatcher. In case of SNMPv3, if security is required, the Message Processing System may pass the PDU to the appropri-

ate security model in the Security Subsystem for encryption and the Security Subsystem may generate an authentication code and insert it into the message header. The processed message is then returned to the Message Processing subsystem which in turn is returned to the Dispatcher. Finally, the Dispatcher maps this message onto the transport layer for transmission.

A swappable SNMP agent is an SNMP agent with a swappable SNMP engine (an SNMP engine with hot-swapping capability). As shown in Figure 5-2, a swappable SNMP engine is composed of a Swap Manager, a Dispatcher, a Message Processing Subsystem, a Security Subsystem, and an Access Control Subsystem. The Swap Manager is the core component in this hot-swapping system. The Swap Manager has access to all S-Components in the SNMP engine, and is responsible for controlling hot-swapping transactions. Some subsystems in the swappable engine have been re-constructed as S-Components and will be detailed in the next section.

The flow of processing SNMP messages inside a swappable SNMP engine is the same as in a traditional SNMP engine. The existence of the Swap Manager is transparent to SNMP applications. However, when the Swap Manager receives a hot-swapping request, it will temporarily interrupt the service of the corresponding S-Components in order to conduct the hot-swapping transaction. Meanwhile, other S-Components and non S-Components in the SNMP engine are still in service. As soon as the hot-swapping transaction is finished (completed or aborted), all the affected S-Components are released by the Swap Manager and back in service.



Note: Shaded modules have been designed as S-Modules

FIGURE 5-2. A Swappable SNMP Agent

5.2.2 S-Components

In the swappable SNMP agent, the following subsystems (or modules) have been targeted and constructed as S-Components. These S-Components are shown in shaded boxes in Figure 5-2.

- In the Message Processing Subsystem, both the V1MP (Message Processing version 1) module and the V3MP (Message Processing version 3) module have been designed and implemented as S-Components.
- In the Security Subsystem, the UserSecurityModel (USM) as a module has been designed and implemented as an S-Component.
- Inside the USM module, the MD5 module, the SHA module as well as the DES module have been designed and implemented as S-Components.

Each S-Component is composed of an S-Proxy and an S-Module as defined in Chapter 4 . For example, the MD5 module (an S-Component) is made of an S-ProxyMD5 (an S-Proxy) and an MD5Module (an S-Module). Only the S-Modules are swappable. As an example, the MD5Module can be replaced with a S-Module of the same type, such as a bug fixed or a function upgraded MD6Module. To make it simple, we only change the identifier of each new S-Module. The algorithm in different versions of an S-Module remains the same.

As discussed in Section 4.2 , an S-Module can have a dependency list which indicates all the other S-Modules that this one depends on. In order to swap in a new S-Module, all the S-Modules in the dependency list must have already been loaded into the system. To test this feature, we had embedded a dependency list into a specific version of the UserSecurityModel (named as USM1 module). This dependency list intently restricts the UserSecurityModel to depend on a certain version of MD5 module (named as MD6 module), SHA module (named as SHA1 module) an DES module (named as DES1 module). So, in order to swap in the USM1 module, all these modules must be swapped in as well.

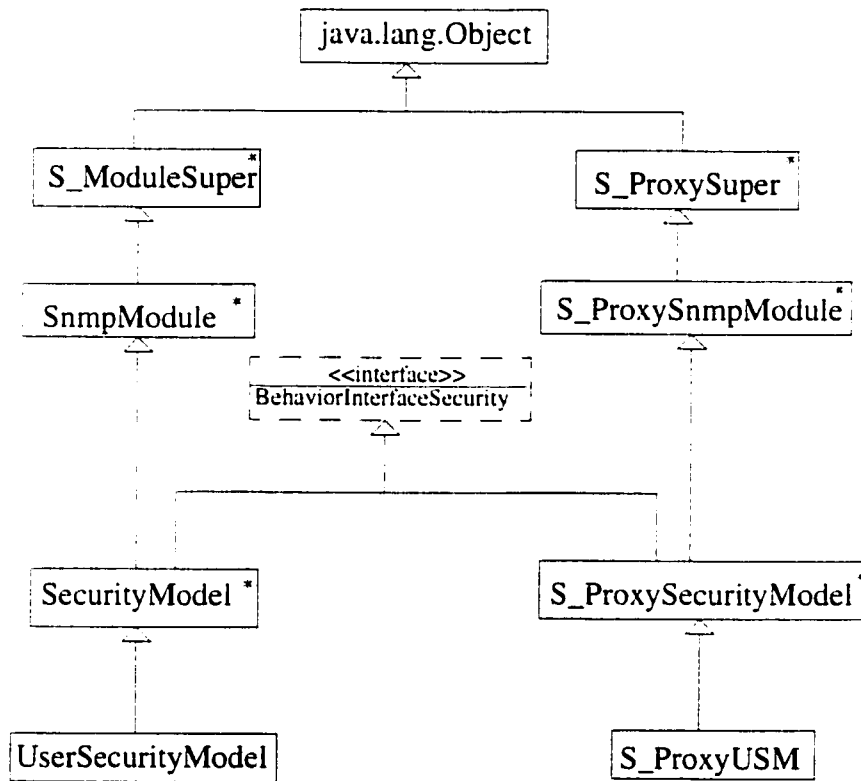
Also, as discussed in Section 4.2 , an S-Module may have some attributes which must be mapped to the new S-Module in order to keep the persistency. For example, the USM module has an attribute which contains the handles to the authentication modules and the encryption modules. This attribute is important, thus must be mapped to the new USM modules. In this application we implemented the attributes mapping by embedding the mapping rules inside the S-Modules, which was discussed in Section 4.2.5.1 .

5.2.3 Class Diagram

The design of the swappable SNMP agent follows the rules defined in Chapter 4 . Classes are structured in a hierarchy to inherit the functionality of other classes and to hide the details of different classes behind a common interface. Abstract classes are used to impose a series of methods for the class that correspond to the implementation.

As an example, Figure 5-3 depicts a class diagram for the User-Based Security Model in the swappable SNMP agent. The diagram shows the relationship between a S-Module class and its S-Proxy class. Java classes and interfaces are designed by following the rules specified in Chapter 4 . The super class of all S-Module classes is the S_ModuleSuper class, and the super class of all S-Proxy classes is the S_ProxySuper class. In Figure 5-3, the SnmpModule class and the SecurityModel class together correspond to the S-Module-TypeX class, the S_ProxySnmpModule class and the S-ProxyUSM class together correspond to the S-ProxyTypeX, the BehaviorInterfaceSecurity class corresponds to the BehaviorInterfaceTypeX class, and the UserSecurityModel class corresponds to the ConcreteS_ModuleTypeX class.

Both the SecurityModel class and the S-ProxySecurityModel class implement the interface BehaviorInterfaceSecurity. New classes of different versions of the User Security Model should also derive from the SecurityModel class, thus inherit all the methods from the S-ModuleSuper and should implement all the methods declared in the BehaviorInterfaceSecurity interface.



Note: shadowed classes are from the hot-swapping framework, classes with * are abstract classes, dashed boxes indicate java interfaces

FIGURE 5-3. Swappable SNMP Agent Class Diagram (User-Based Security Model)

5.3 Tests and Results

The swappable SNMP agent application is set up on two separate machines. One represents an SNMP manager (hereby called manager) listening at port 10000, and the other represents a swappable SNMP agent (hereby called agent) listening at port 161 (Figure 5-4).

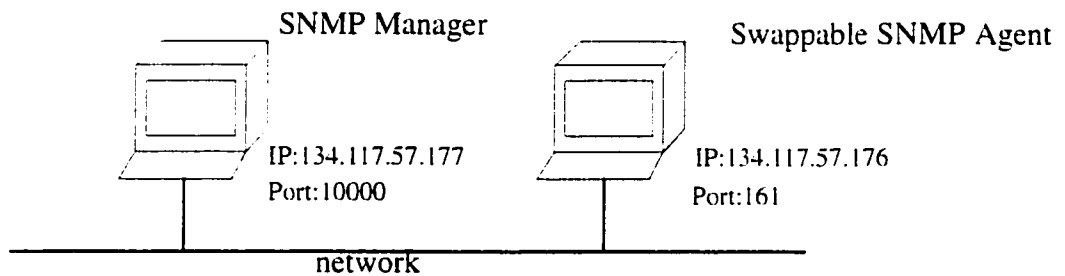


FIGURE 5-4. Test System Environment

For the initial contact, the manager sent a broadcasting message in the SNMPv3 format (with no security) to the agent. When the agent gets this initial message, it retrieves the information about the manager, such as the IP address, the engine ID, engine boots, engine time, etc. from the message header. In order to support SNMPv3 format messaging, both the manager and the agent should be configured to the same user list which will be used by the USM security module. The USM module can configure users into three categories: no security, with authentication (MD5/SHA), with both authentication (MD5/SHA) and encryption (DES).

After the initialization and configuration, the manager may send SNMP request messages (in SNMP v1 or v3 format) to the agent giving the object ID (OID). The agent responds to

the request and processes security check on the incoming message. If it passes, the agent then gets the value of the requested OID from the local MIB implementation, encapsulates the data in an SNMP response message and sends it back to the manager.

When the agent starts up, the Swap Manager which sits inside the agent starts running in a separate thread. The Swap Manager has a user interface which displays a list of the S-Components currently in the agent, a list of new S-Modules which are available to be swapped into the agent, and a list of new S-Modules selected from the new S-Module list and will be swapped into the system within one transaction. The Swap Manager controls the hot-swapping transaction by committing the transaction or aborting it at any stage. The transaction steps are recorded and displayed on the UI.

The tests on hot-swapping transaction take place within the agent while it is processing messages from/to the manager. Several cases have been tested, these include:

- hot-swapping a single independent S-Module, for example, replacing the MD5 module with the MD6 module;
- hot-swapping a group of independent S-Modules, for example, replacing the MD5 module and the DES module together with the MD6 module and DES1 module;
- hot-swapping an S-Module which has a dependency list of other S-Modules, for example, replacing the USM module with the USM1 model (which depends on MD6, DES1, and SHA1).

Two timers are used to evaluate the hot-swapping transaction. Timer1 tells how long a hot-swapping transaction takes place. It records the time period from a transaction request is issued to the Swap Manager to the point when the system is back to full function with the

new S-Modules. Timer2 records the actual period which the system has been affected. During this time, parts of the system (the selected S-Modules) have been blocked and temporarily out of service while the other parts are always in service.

The activities which relate to a hot-swapping transaction are as follows:

1. The Swap Manager gets a hot-swapping request from its UI input. It reads in all the selected new S-Modules which are requested to be swapped in within a single transaction. It also starts the Timer1.
2. The Swap Manager checks with each selected new S-Module to see if there are any other S-Modules that this S-Module depends on. If it has, the Swap Manager checks to see if these S-Modules are in the system already. If the S-Modules that this new S-Module depends on have not been loaded into the system yet, the Swap Manager checks with the list of all available new S-Modules. If the S-Modules that this S-Module depends on are in the list, then they are added to the selected S-Module list, and will be loaded within the same transaction but right before loading this new S-Module. Otherwise, this transaction can not go on and will be aborted by the Swap Manager.
3. After reorganizing the selected new S-Module list, the Swap Manager asks the corresponding S-Proxies to prepare swapping, which in turn block all new calls to the current S-Modules. The Timer2 is started at this time. If the current S-Module is in the swappable state, the Swap Manager will get the attributes from the current S-Module and map them into the corresponding new S-Module.
4. When all the related S-Proxies have prepared for the swapping and all the attributes have been mapped to the new S-Modules, the Swap Manager will swap in the new S-

Modules and swap out the old ones. If any of the S-Modules failed the preparation and attributes mapping, the whole transaction fails too.

5. After the swapping, the Swap Manager asks all the corresponding S-Proxies to release the blocked calls, and stops the Timer2. The S-Modules are ready in service again.
6. The Swap Manager makes some clean up to be ready for the next transaction and stops the Timer1. This hot-swapping transaction is completed.

The test results show that the SNMPv3 agent was not shut down due to the hot-swapping transaction. During the hot-swapping transaction some parts of the message processing (depends on which S-Module was chosen to be swapped) at the agent side had been momentarily blocked by the swap manager, other parts of the agent was still running and processing messages. Therefore the communication between the manager and the agent was not terminated by the hot-swapping transaction. The SNMP application at the manager side (which kept sending SNMP requests) did not notice the transaction. The test shows that the hot-swapping transaction time (Timer1) at the agent side usually is between tens of milliseconds to a couple of hundred milliseconds, while the actual system affected time (Timer2) is under 30ms. In this application the temporary interruption of the message processing due to the hot-swapping transaction is a minor performance issue, comparing with the network delay which may happen for seconds. The standard time-out value is 120 second, according to the RFC2571 [15].

The tests also demonstrate that the whole hot-swapping transaction time and the agent affected time are determined by the following facts:

- the number of S-Modules to be hot-swapped within one transaction

- the number of S-Modules in the S-Module's dependent list
- the number of S-Components in the agent

However, due to the limitation of this application, it did not demonstrate the dynamic messaging function of an S-Module. Also, since the test is focused on the design and implementation of S-Modules which follows the rules specified in the previous chapter, it is not a full test on all the functionality of the Swap Manager. A test on the whole services of the Swap Manager, such as the listening service, the security service, the two-phase commit transaction etc., and a test on the protocol for code remote delivery and dynamic object instantiating are under research by colleague Gang Ao.

Chapter 6 Conclusions, Contributions and Suggestions for Future Research

6.1 Conclusions

In this thesis we have proposed a new infrastructure for the software hot-swapping applications. The infrastructure and its key components are described in Chapter 2 . Several design approaches have been investigated in Chapter 3 . and the Proxy Pattern approach has been selected for the further deployment (Chapter 4). The design of the S-Modules and the hot-swapping framework has been tested successfully by applying them to a real application in the network management domain. The test results of the swappable SNMPv3 agent demonstrate that dynamic upgrading one or a group of S-Modules within the swappable SNMPv3 agent only causes momentary interruption of the message processing, the agent is still active and in service while the hot-swapping transaction takes place (Chapter 5).

It has been demonstrated that a hot-swapping software system can be achieved by following the technology and methodology described in this thesis. The software upgrading issue can be solved by a common technology, instead of a system/program specific technology.

Today's computer and software technology are efficient enough to handle on-line real-time software upgrading issue.

6.2 Contributions

This thesis has the following contributions:

1. A software hot-swapping infrastructure has been designed to achieve the goal of software hot-swapping.
2. Several approaches for designing the S-Module have been investigated and analyzed. Among those approaches, the Proxy Pattern approach has been promoted to a detailed design, which leads to a design of the hot-swapping framework.
3. An SNMPv3 agent is re-constructed according to the design rules described in the hot-swapping framework, and the test on dynamic upgrading one or more S-Modules of the agent has been accomplished successfully.

6.3 Suggestions for Future Research

This thesis is just a starting point in the research of software hot-swapping technology. There are a number of directions that can be considered by people who are interested in extending this thesis's work:

1. As mentioned in Section 2.2 on page 6, several other important features in the hot-swapping infrastructure, such as a protocol for code migration, the dynamic object instantiation, a detailed design on the complex hot-swapping transaction, should be

developed and integrated with this thesis work to achieve the goal of a complete hot-swapping activity.

2. In this thesis we have investigated several design patterns, such as Proxy Pattern, Observer Pattern, Mediator Pattern, for their employment in the software hot-swapping technique. Other design patterns, such as the Facade Pattern, may also be employed to achieve the goal of software hot-swapping. More investigation and comparison could be done in the design of hot-swapping framework.
3. The Proxy Pattern approach has been developed and implemented in this thesis. The further evaluation of this approach, especially on the performance assessment, could reveal more advantages and disadvantages of the hot-swapping framework, as well as the infrastructure which has been described in Chapter 2 .
4. Although in this thesis we have focussed on an application in the network management software entity, the SNMPv3 agent, the research on software hot-swapping is widely applicable within the Software Engineering domain. Other more complicated applications may assess more features which could not be tested by this single application.

References

- [1] Gang Ao, "Software Hot-swapping Technique (draft)," Technical Report SCE-98-11, Systems and Computer Engineering, Carleton University, Ottawa Canada, November 1998.
- [2] N.Feng, G.Ao, T.White and B.Pagurek. "Software Hot-swapping Technology Design." Technical Report SCE-99-04, Systems and Computer Engineering, Carleton University, Ottawa, Canada. June 1999.
- [3] E.Gamma, R. Helm, R.Johnson, and J. Vlissides. "Design Patterns: Elements of Reusable Object-Oriented Software." Addison-Wesley Publishing Company, 1995.
- [4] Tim Lindholm and Frank Yellin. "The Java Virtual Machine Specification." Addison-Wesley, 1997.
- [5] Laura Lemay and Charles L. Perkin, "Teach Yourself Java 1.1 in 21 Days," Sams.net Publishing, 1997.
- [6] Alex Blewitt, "Use Dynamic Messaging in Java," available at URL: <http://www.javaworld.com/javaworld/javatips/jw-javatip71.html>.
- [7] "Java™ Development Kit 1.1." available at URL: <http://java.sun.com/products/jdk1.1>.
- [8] Chuck McManis, "Take an In-depth Look at the Java Reflection API," available at URL: <http://www.javaworld.com/javaworld/jw-09-1997/jw-09-indepth.html>.

- [9] Bill Venners, "Design with Run-time Class Information," available at URL: <http://javaworld.com/javaworld/jw-02-1999/jw-02-techniques.html>.
- [10] "SNMP Version 3 (SNMPv3)", available at URL: <http://www.ibr.cs.tu-bs.de?ietf/snmpv3/>.
- [11] William Stallings. "SNMP, SNMPv2, and CMIP, the Practical Guild to Network Management Standards," Addison-Wesley, 1993.
- [12] "Modular SNMP Project," available at URL: <http://www.teleinfo.uqam.ca/snmp/>.
- [13] Omar Cherkaoui, Nathalie Rico. "SNMPv3 Can Still Be Simple?." IEEE/IFIP Network Operations and Management Symposium (NOMS'99), Boston, MA, May 1999.
- [14] William Stallings. "SNMPv3: A Security Enhancement for SNMP." IEEE Communication Surveys, Fourth Quarter 1998, Vol.1 No.1.
- [15] D.Harrington, R.Presuhn, B.Wijnen. "An Architecture for Describing SNMP Management Frameworks." RFC2571, May 1999.
- [16] J.Case, D.Harrington, R.presuhn, B.Wijnen. "Message Processing and Dispatching for the Simple Network Management Protocol (SNMP)," RFC2572, May 1999.
- [17] D.Levi, P.Meyer, B.Stewart, "SNMPv3 Applications." RFC2573, April 1999.
- [18] U.Blumenthal, B.Wijnen, "The User-Based Security Model for Version 3 of the Simple Network Management Protocol (SNMPv3)," RFC2574, April 1999.
- [19] B.Wijnen, R.Presuhn, K.McCloghrie, "View-based Access Control Model for the Simple Network Management Protocol (SNMP)," RFC2575, April 1999.