

SOFTWARE HOT-SWAPPING TECHNIQUES FOR UPGRADING MISSION CRITICAL APPLICATIONS ON THE FLY

by

Gang Ao, B.Eng.

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

Master of Engineering

Ottawa-Carleton Institute for Electrical Engineering
Faculty of Engineering

Department of Systems and Computer Engineering
Carleton University

Ottawa, Ontario
CANADA, K1S 5B6

February, 2000

(c) Copyright 2000, Gang Ao

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

**SOFTWARE HOT-SWAPPING TECHNIQUES FOR
UPGRADING MISSION CRITICAL APPLICATIONS
ON THE FLY**

submitted by Gang Ao, B. Eng in partial fulfillment of the
requirements for the degree of Master of Engineering



Thesis supervisor
Professor Bernard Pagurek



Thesis supervisor
Tony White



Chair, Department of Systems and Computer Engineering
Professor Rafik A. Goubran

ABSTRACT

With the increasing demand for long running and highly available distributed services, the ability to upgrade without taking down the operations of the mission critical software deployed in the field is of great significance to today's fast growing telecommunication and information industry. It is noted that none of the existing software maintenance approaches has provided a feasible solution to fulfil this goal. This thesis therefore proposes a software hot swapping technique to accomplish a generic and robust infrastructure at the initial design stage to accommodate the need of future software maintenance with minimum system service disruption. A description and thorough discussion of issues concerning the software swappability as well as software hot swapping infrastructure consisting of S-module, S-manager and S-proxy, is systematically presented for the purpose of dynamic reconfiguration.

The software hot-swapping infrastructure provides a auto-managed framework where subset of the software can be reconfigured on a module by module basis while overall software execution consistency is maintained. The developed hot swap transaction algorithm places special emphasis on minimizing the interference to the rest of the system and suits well for the applications with zero-down time requirements. The research demonstrates that the object-oriented paradigm combined with mobile code technology provides a solid foundation for the software hot-swapping technique. By applying the software hot-swapping technique, the maintenance cost for software product, especially for distributed, mission critical application, can be significantly reduced.

摘要

隨著網絡技術的飛速發展，越來越多的業務需要長期不間斷地提供服務。特別是在通訊和信息工業領域，能夠在線實現軟件升級而不中斷其業務具有非常重要的意義。然而，目前所有軟件維護的技術都無法實現這個目標。針對這個難題，本論文提出了一種全新的在線置換軟件的技術，在設計軟件的一開始就建立一個通用可靠的系統，以滿足日後軟件升級維護而不中斷其運行服務的需要。本論文對於有關軟件可置換性的問題進行了詳細的論述，並且系統介紹了為動態管理軟件而建立的軟件在線置換系統(包括可置換軟件單元、升級管理器和置換單元接駁器)。

本軟件在線置換系統提供了一個自動管理的軟件系統，使得應用軟件可以按可置換單元為基礎進行更新，並保持軟件服務的連續性。特別是我們提供了一種在線置換軟件的操作機製，可以儘量地減少對系統運行的干擾，尤其適用於運行不可中斷的軟件。本研究結果表明，面嚮目標的軟件系統和可移動軟件技術相結合，提供了在線置換軟件技術的基礎。通過這種在線實現軟件升級的技術，軟件(尤其是分散式、運行不可中斷的軟件)的維護費用可以大大降低。

ACKNOWLEDGEMENT

First I would like to express my gratitude to my two supervisors, Professor B. Pagurek and Dr. T. White, for their guidance and patience throughout the preparation of this thesis. I would also like to thank Dr. Andrzej Bieszczad who originally suggested the research topic. The financial assistance from Nortel Networks Corporation is sincerely appreciated.

I would like to take this opportunity to thank all my good friends who always give me encouragement and support no matter where I am. A special thank to my best friend, Yi Wang, for the extremely good understanding and love. Without friendship, my life would be meaningless.

Finally, I want to thank my parents, my dear brother and sister. From the earliest days, they have encouraged me to strive for excellence and provided overwhelming understanding, unconditional love and support that have enabled me to achieve it.

Table Of Contents

Abstract	iii
Acknowledgment	iv
List of Figures	viii

CHAPTER 1.0 INTRODUCTION

1.1	Research Motivation.....	1
1.1.1	Software Maintenance is Costly	2
1.1.2	The existing approaches for software maintenance.....	7
1.2	Thesis Objective	11
1.3	Thesis Organization.....	13

CHAPTER 2.0 THE FOUNDATION OF SOFTWARE HOT-SWAPPING

2.1	The basic requirements.....	15
2.1.1	Module Based Software Structure	16
2.1.2	Dynamic Extensibility	18
2.1.3	Managed Interface Access.....	18
2.1.4	State Sensitive.....	19
2.1.5	Delay Sensitive	20
2.1.6	Minimizing Side-effect.....	20
2.1.7	Service Transparency.....	21
2.1.8	Security.....	21
2.1.9	Code Mobility.....	22
2.1.10	OS Support	22
2.2	Object Oriented Paradigm as the Foundation.....	23
2.3	Supports from Java Architecture	25
2.3.1	An object-oriented paradigm	25
2.3.2	Dynamic linking and dynamic extension	27
2.3.3	Synchronization service.....	28
2.3.4	Platform independence	29
2.3.5	Security	30
2.3.6	Network mobility.....	35
2.3.7	Run time introspection.....	35

2.3.8	Java's Drawback.....	36
2.3.9	Alternatives.....	37
2.4	Potential solutions for software hot-swapping technique.....	38
2.4.1	Solutions at Operating System Level	38
2.4.2	Solutions at system level	40
2.4.3	Solutions at application level.....	41
2.4.4	The research assumption.....	43
2.5	Conclusion.....	44

CHAPTER 3.0 SOFTWARE HOT-SWAPPING ARCHITECTURE

3.1	An overview	45
3.2	The specification of S-modules	47
3.2.1	The characteristics and interface of an S-module.....	48
3.2.2	Prepare for S-module	53
3.2.3	Negative effects of OO	55
3.3	S-proxy	56
3.3.1	S-proxy for referential problem	58
3.3.2	Functional modification and extension.....	59
3.4	Swap manager and its services	62
3.4.1	The services of swap manager.....	62
3.5	Conclusions	66

CHAPTER 4.0 THE SOFTWARE HOT-SWAPPING TRANSACTION

4.1	An overview of transaction concept.....	67
4.1.1	The notation and the ACID properties.....	67
4.1.2	Approaches for the state consistency of an object.....	68
4.2	The transaction in the hot-swapping architecture.....	69
4.2.1	The S-application transaction	70
4.2.2	The swap transaction in the hot-swap architecture.....	70
4.2.3	The state of an S-module and its checkpoint.....	72
4.3	The state machine of an S-module	80
4.4	The state machine of an S-proxy	82
4.5	S-manager in swap transaction	85
4.5.1	The state machine of the Swap Manager.....	85
4.6	The two-phase commit transaction model.....	87

CHAPTER 5.0 IMPLEMENTATION AND APPLICATION

5.1	The implementation of the software hot-swapping architecture	93
5.1.1	Use case diagram	93
5.1.2	Interface for the S-proxy	96
5.1.3	Sequence diagram	99
5.2	Application of software hot-swapping architecture	102
5.2.1	An experimental application	103

CHAPTER 6.0 CONCLUSIONS

6.1	Summary	117
6.2	Conclusions	119
6.3	Directions for Future Work	120

List of Figures

Fig. 1.1: Cost of each phase of software lifecycle.....	3
Fig. 2.1: Module based software	17
Fig. 2.2: Inter-module communication Middleman	18
Fig. 2.3: The Java Virtual Machine	25
Fig. 2.4: Swap an object inside a Java Virtual Machine	40
Fig. 2.5: Different proposals for composing S-application software	41
Fig. 3.1: A server application in a distributed environment	45
Fig. 3.2: The software hot-swapping architecture	46
Fig. 3.3: Referential update	57
Fig. 3.4: The problem of reference propagating.....	57
Fig. 3.5: Proxy approach for referential problem.....	58
Fig. 3.6: Functional extension problem.....	60
Fig. 3.7: NewService interface for an S-proxy.....	61
Fig. 4.1: A swap transaction.....	71
Fig. 4.2: Identify S-application Transaction and Swap Transaction	78
Fig. 4.3: The state machine of an S-module	81
Fig. 4.4: The state machine of an S-proxy.....	83
Fig. 4.5: The state machine of the Swap Manager	86
Fig. 4.6: The two-phase commit transaction model	92
Fig. 5.1: Use case diagram for the software hot-swapping architecture	94
Fig. 5.2: Main Classes Diagram for the Software Hot-swapping Architecture.....	95
Fig. 5.3: The important roles of an S-proxy	98
Fig. 5.4: Sequence Diagram for the case of swapping an S-module	99
Fig. 5.5: The structure of a swappable SNMP Agent.....	102
Fig. 5.6: An experimental application of our hot-swapping technique	108
Fig. 5.7: Swap S-modules with dependent relationship	110
Fig. 5.8: Administrator panel.....	111
Fig. 5.9: Application Client panel	112
Fig. 5.10: Application server and its swap manager	113
Fig. 5.11: The result of swapping S-module S1 and S-module S3.....	114
Fig. 5.12: The print out of swapping S-module S1 and S-module S3.....	115
Fig. 5.13: The result of aborting a swap transaction	116

CHAPTER 1.0 INTRODUCTION

1.1 Research Motivation

This thesis research is fundamentally motivated by the challenge of upgrading distributed, mission critical software while it is in operation without taking down its service. First of all, there is an increasing demand for a flexible and cost-effective solution for the software dynamic reconfiguration from the telecommunication and information technology industry. Secondly, there have not been any viable solutions that adequately address the industry demand.

The term, "hot swap", which was originally used for the replacement of a hardware device while the system remains in operation [1], has been used in this thesis for the replacement of a part of software programs while the overall program remains in operation. By achieving this goal, the ability to maintain distributed, mission critical software can be significantly improved and the maintenance cost can be significantly reduced.

1.1.1 Software Maintenance is Costly

With communication networks developing and evolving at a rapid pace, distributed computing and distributed embedded systems have become more and more widespread in commercial, industrial and research establishments. Notably, functionality of communication equipment is largely performed by software. In general, software often needs to be changed for the following well-known reasons:

- ◆ Software is prone to human error, thus it can not be perfect.
- ◆ Software is a model of reality, and as that reality changes, the software must adapt.
- ◆ The functionality of the product has to be extended beyond what is feasible in terms of the original design.
- ◆ Software is easier to change than hardware. On the one hand, successful software survives well beyond the lifetime of the hardware for which it was written. On the other hand, more appropriate hardware components become available while the software is still viable. In general, the software will have to be modified to some extent in order to run on the new hardware.

Approximate relative costs of the phases of the software life cycle

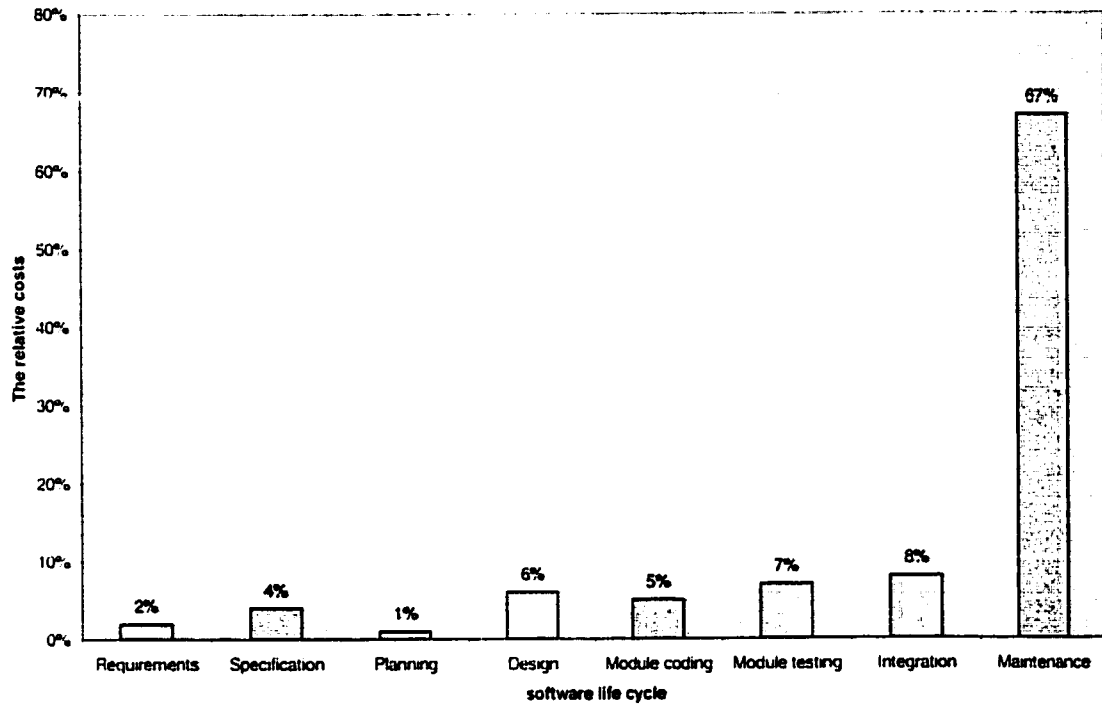


Fig. 1.1: Cost of each phase of software lifecycle

As a matter of fact, studies have indicated that about two-thirds of the total software effort is devoted to maintenance and upgrade, and more money is spent on maintenance and upgrade than on all other software activities combined [2, 3]. Fig.1.1 illustrates the approximate percentage of time/money spent on each phase of the software life cycle.

In distributed systems, it is extremely common that the system is quite complex and software is distributed across multiple networks/system domains. Therefore distributed software is unlikely to be fault-free and ongoing bug fixes and functional enhancement are inevitable. On the other hand, distributed systems need to evolve constantly as market demands, technology, and the application environment are changing constantly. Hence one of the major challenges for designing distributed systems is to possess the ability to accommodate evolving changes, particularly with respect to the management of changes, as many system functions may require later upgrades. Those upgrades normally involve bug fixes, modification of an existing function in the system or an extension of system features by introducing new functions.

In general, evolutionary changes are not easy to predict as they can neither be clearly defined nor identified at the system design stage. Due to time to market pressures, a product needs to be deployed before enhanced versions become available. Therefore maintenance is particularly difficult for communication network applications that are widely distributed across heterogeneous domains.

In summary, the extra difficulty stems from the following sources:

Firstly, many distributed applications are distributed so broadly that it is difficult, sometimes even impossible, to maintain them on site. Consequently, it is important to be able to maintain these applications remotely.

Secondly, mission critical applications are required to have zero or close to zero down time. As a result, in many cases, it is extremely important not to have to take the system down for software upgrading and/or recompilation. Besides calculating the maintenance cost of distributed, mission critical applications, one must take the cost of system interruption into consideration.

Thirdly, the scheme for maintaining software on the fly has to be robust, minimizing the disruption to the applications and leaving the applications in a consistent state. This requirement is the most challenging one, as the applications must run continuously even though there are frequent needs for bug fixes or functional upgrades

Fourthly, maintaining software on the fly should be transparent to the users of the application and it should not expect the cooperation from the user side during the process of maintenance.

For the above reasons, many of the applications that are distributed, complex, and mission critical require special consideration in their maintenance approaches. Such applications can be found in network management, communication switches, process controllers, and security applications, etc. For instance, upgrading software on telecommunications equipment deployed in the field is often a major headache, as telecommunication carriers demand that their equipment can tolerate no more than a few minutes of downtime per year. In many cases, telecommunications equipment makers are mandated by government regulations to provide zero or close to zero downtime.

Therefore, there is a real incentive in the communication industry to resolve the maintenance problem with a sound underlying infrastructure under the pressure of very stringent requirements for system disruption.

We believe that the key to reducing software maintenance costs is to resolve the problem at the initial design stage. That is, the system software should be designed such that it is sufficiently flexible to permit incremental changes when necessary. It is required that software design should, as far as feasible, take future dynamic reconfiguration into account and provision all possible upgrades.

1.1.2 The existing approaches for software maintenance

In practice, the following approaches are presently being used in the industry for software upgrade. The pros and cons of each individual approach are listed as follows.

1.1.2.1 Re-installing new software

Taking down the device and re-installing new version of software is the most popular way of upgrading software. Its advantage is that the software can be changed without much constraint. The disadvantage is that the service is interrupted. Sometimes the whole system has to be shut down for a small change of the code. It often takes a significant amount of time and effort to take down the device, install new software, and reboot all the devices. Clearly mission critical applications can not afford this type of interruption due to their close-to-zero downtime or zero downtime requirements.

1.1.2.2 Patching

According to the definition [1], a patch (sometimes called a "fix") is a quick-repair job for a piece of program. Problems (called bugs) in a program will almost invariably be found. A patch is usually developed and distributed as a replacement for or an insertion into compiled code (that is, in a binary file or object module) and some memory space is left for this kind of insertion. In larger operating systems, a special program is provided to manage and keep track of the installation of patches.

The advantage of patching is that it reduces the time to upgrade software. The disadvantage is that the device still needs to be rebooted thus causing service interruption. There are also limitations in program modification plus the uncertainty as to how much spare memory space to provision in the first place, which could be either a waste or insufficient for the code insertion.

1.1.2.3 Redundant device

A redundant device stands by and traces the working device. Software upgrades can be performed on a spare device. Then, the spare device takes over the job from the working device. Traditionally telecommunication equipment makers have relied on network and/or equipment redundancies to facilitate upgrades, which not only allow smooth upgrades but also support fault tolerance. This approach belongs to hardware hot-swap.

However, this traditional method also has its limitations. It is more suited to centralized large and expensive equipment where space and cost are less of a concern than it is to a system distributed over the network. Another disadvantage of this approach is that some system transactions may be lost during device switching. Also, some system data may not be share-able between the working and redundant device because they are associated with the installation and configuration of different devices and different versions of software.

1.1.2.4 Parallel processor

Similar with redundant device approach, a pair of parallel processors can be used for upgrading software. One processor works on the old version of software while a parallel processor works on the new version of software. They share data and files through shared memory, so one processor can be switched to another one and continuously work on the same data. Its advantage is that the switch procedure between parallel processors can be seamless and there is no need for extra device. Its disadvantage is that it requires a special operating system and it is complicated and not feasible for many embedded applications. There are also limitations on the type of software change. For example, the two processors have to work exclusively on shared data and files.

1.1.2.5 Dynamic object technology

This is an existing approach that allows for software changes on the fly. Common Lisp Object System (CLOS) takes advantage of the Lisp programming language which can be modified while the program is running without necessarily having access to, or knowledge of, those parts of the application which are unaffected by the change [4, 5, 6].

Its advantage is that programs can be developed and delivered more easily, because only the broken part needs to be refined while the application is still running. Mission-critical applications can be maintained more easily, because the programs themselves can be simpler, more robust and easier to develop and maintain. Its disadvantage is that reflective languages including Lisp are not very popular in industry application. Moreover, the execution speed of Lisp is relatively slow.

1.1.2.6 Mobile Agent Technology

Mobile agent technology demonstrates some success for upgrading software dynamically. Through the agent, certain existing applications can be upgraded at run time without interrupting its service. A research [7] demonstrates that DPI protocol can be used for a mobile agent to dynamically extend SNMP MIB and the RDPI protocol can be used for a mobile agent to issue management requests to the SNMP agent, in this way it actually can upgrade SNMP MIB at run time. However, this technology has a rather limited application scope in that it did not provide a general framework for software reconfiguration.

In view of the foregoing, it is clear that none of the existing software upgrade approaches is sufficient for distributed mission critical software maintenance. We believe that it is absolutely necessary to develop a new technique to support low cost and high efficiency software maintenance with minimum service disruption. With this technique, mission critical applications will have a sound underlying software infrastructure to achieve this goal.

With the arrival of the object-oriented programming language Java which provides network mobility, network security, platform independence as well as dynamic linking and dynamic extension features etc., it is finally possible of developing a software technique that is inherently suitable for accommodating future upgrade at the initial design stage. This new technique, which we call software hot-swapping, is aimed at enabling the system administrator to upgrade or replace the software remotely while the whole software system remains in operation.

1.2 Thesis Objective

Therefore, this thesis research is focused on developing a technique, the hot swapping technique, allowing on-line program maintenance without taking down its service. The objective of this thesis is to systematically present the software hot-swapping technique and its infrastructure.

With our software hot-swapping infrastructure, software executing in a managed element in a network can be upgraded without disrupting the existing execution environment and state of operations or relying on an external redundant back-up unit, i.e., the hot-swap of software on a module-by-module basis to maintain software on the fly. The research is to demonstrate that the object-oriented paradigm combined with mobile code technology can provide the foundation for software hot-swapping technique. By applying the software hot-swapping technique, the maintenance cost for software products, especially for those distributed, mission critical applications can be significantly reduced.

Accordingly, this thesis is aimed at the following:

- ❶ Having assessed the pros and cons of the existing software maintenance approaches, a set of principles on which the swappability of distributed and mission critical software is based are to be proposed. To this end, software modularity is to be studied and software hot swapping foundation is to be presented.
- ❷ To systemically present a hot swapping infrastructure to manage the dynamic software upgrade on demand automatically. And to define all the major components and their roles and services in the infrastructure.

-
- ③ To develop a generic strategy on maintaining the state integrity of an application during swap transactions. And to present a two-phase-commit transaction model in order to ensure the robustness and the service continuity of the application.
 - ④ To analyze various practical scenarios that the hot-swapping technique may encounter and provide comprehensive solutions and guidelines. To implement the hot-swapping infrastructure and prove its concept and usages in real applications.

1.3 Thesis Organization

The rest of thesis is organized as follows.

Chapter 2 will start with the discussion of the requirements and principles for hot-swappable software. With the introduction of the object-oriented paradigm and Java features, the foundation of software hot swap techniques and its challenging issues will be presented.

Chapter 3 will introduce the software hot-swapping architecture and its major components including the S-module, S-manager and S-proxy. A detailed description of the roles, characteristics and services of each component will be provided.

Chapter 4 will analyze the hot-swap transaction within the software hot-swapping architecture. The two-phase commit transaction model will be introduced and strategies to tackle various challenging issues in hot-swapping transactions will be detailed.

Chapter 5 will describe the implementation and testing of the proposed software hot-swapping architecture.

Chapter 6 will summarize the thesis contribution, draw conclusions and recommend future work.

CHAPTER 2.0 THE FOUNDATION OF SOFTWARE HOT-SWAPPING

To reduce the cost of software maintenance and to accommodate future reconfiguration at the initial design stage, it is important to clearly understand the basic requirements and principles for swappable software. It is based on these principles that software hot-swapping foundation is established to tackle various challenging issues in upgrading mission critical software on the fly without taking down its services.

2.1 The basic requirements

Not any arbitrary application can be upgraded on the fly. On the contrary, most existing software is machine dependent and can not be dynamically modified. Swappable software requires an architecture that can manage and facilitate swapping operation while sustaining most of its services. Moreover, the applications applying hot-swapping technique (hereafter called **S-applications** in this thesis) should be constructed according to certain generic principles that can be realized in a simple, robust and reliable fashion.

The following details the requirements for swappable software coming out of this thesis research. Some of them are functional requirements, which means that the goal of software hot-swapping can not be achieved without it. Other requirements are non-fundamental requirements, which means although desirable, you may swap software without necessarily satisfying these requirements. However, non-functional requirements are also very important features of the technique.

2.1.1 Module Based Software Structure

Swappable software should have a modular structure that consists of one or more swappable modules. It must be configured into a set of swappable modules based on certain criteria so that it can be replaced/changed in a module-by-module fashion instead of altering the entire program all at once, which would surely cause service interruption. The modular structure is based on the decomposition criteria known as information hiding. According to this principle [8], system details that are likely to change independently should be the secrets of separate modules; each data structure is used in only one module; it may be directly accessible within the module but not from outside the module. The only way to access information stored in a module's data structure from outside is through those interfaces provided by that module. Therefore, any change on one independent module will not affect other modules in the program.

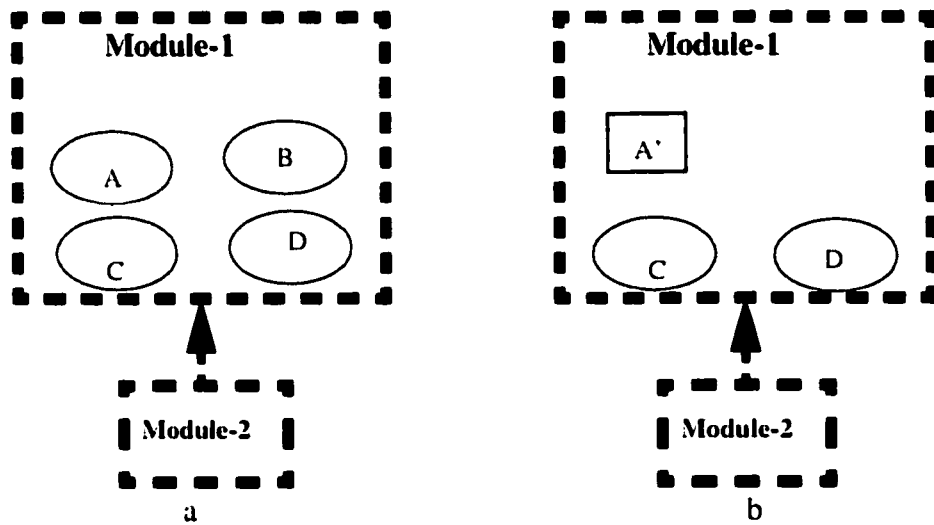


Fig. 2.1: Module based software

For example, in Fig. 2.1a, Module-1 has some internal information like A, B, C and D and Module-2 can only access this information through the interface of Module-1. If A is changed into A' and B is removed from Module-1(Fig. 2.1b), these changes remain inside of Module-1 and will not affect Module-2. In another word, Module-1 gets upgraded without affecting its cooperation with Module-2.

It should be pointed out that module based software is not a functional requirement for software hot-swapping, because some interpreted programming languages like Lisp and Prolog can be upgraded on-the-fly without following this rule. However, modular structure is the main stream of software application.

2.1.2 Dynamic Extensibility

Keep in mind that swappable software needs to be modified at run time and re-compilation at run time without interrupting service is not possible. Software upgrading will involve some changes of interface and functionality, and the software hot-swapping architecture should be able to manage and support interface extension and interface change dynamically.

2.1.3 Managed Interface Access

Furthering the modular information hiding principle, the inter-module communication of the application has to be controlled to retain the inter-module relationship after one or multiple module have been swapped.

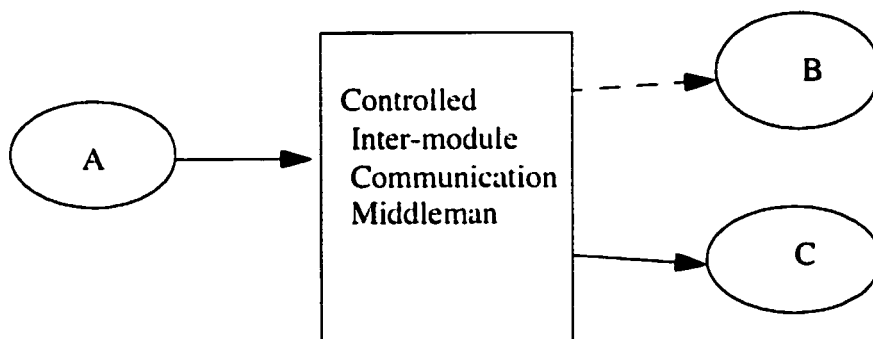


Fig. 2.2: Inter-module communication Middleman

For instance, as shown in Fig. 2.2, if Module A and Module B have a direct communication interface, when B is replaced by C, A's communication channel should be switched to C automatically from B. The inter-module communication can be controlled through a middleman layer so that the relationship between those modules can be changed at runtime. In another words, the software hot-swapping infrastructure has to provide some types of middleman services to manage the inter-dependent modules at runtime.

2.1.4 State Sensitive

The hot swappable software is state sensitive. The mechanisms involved in a hot-swap must keep the integrity and continuity of the application while the swap is taking place. It is important that as the application is being upgraded on the fly, its state has to be preserved for all the active transactions. Otherwise, the integrity and continuity of the application can not be guaranteed. In other words, whether or not the swap operation is successful, valuable system data should not be lost in between.

To meet this requirement, the application services activities must be synchronized with the swap activities in order to manage the state of the application. Since this mechanism involves many low-level and error-prone activities, it has to be done in an automatic and robust fashion.

2.1.5 Delay Sensitive

Many mission critical software applications are real time applications. One of the main reasons why an application needs hot-swapping is because that its services have to be available to its users at all times. Therefore a time constraint is necessary to prevent the hot-swap activity from violating the availability of the application. The hot-swap operation itself should be able to work under a time constraint in order to minimize the interruption for the S-application. In other words, the hot-swapping technique should have a very efficient mechanism to minimize its interruption to the S-application and satisfy the time constraint.

2.1.6 Minizing Side-effect

The subset of the software modules undergoing the hot swap should not interfere much with the execution of the rest of the system. As described in the section 2.1.1, any application that applies the hot-swapping technique has to be configured into a certain module-based structure. In order to be practical, the hot-swapping technique should be simple, efficient, robust and scalable. This modular structure should not introduce too much overhead into the application that would affect its real time performance. Obviously, there is no reason to sacrifice too much efficiency of an S-application merely for a possible evolution that might not occur very often.

As we know, the inter-module communication of an application is normally through local method invocation. However, distributed objects such as Jini components communicate through RMI or CORBA objects communicate through an ORB both of which have a performance penalty during the procedure of marshalling and un-marshalling parameters.

Side effects that the software hot-swapping technique may bring to an S-application also include system resource occupation. The software hot-swapping infrastructure should not be too "bulky". Otherwise, its scalability and real time response would be very limited.

2.1.7 Service Transparency

The hot-swap operation should be transparent to client applications and it can not expect any cooperation from its client side during the process of hot-swapping. Also, clients should not suffer from any notable service degradation overall.

2.1.8 Security

Security is an important issue to consider in the hot-swap operation. Mission critical applications often suffer from malicious attacks from hackers and viruses. Without some guarantee of the security of the system and application, the hot-swapping technique will not have any practical significance. Obviously, all the network security schemes can be used to safeguard the hot swap transactions.

2.1.9 Code Mobility

Remote maintenance of distributed applications requires code mobility over the network. With code mobility, new versions of software modules can be prepared remotely and then sent to the target network node to replace stale versions.

Ideally, the mobile code should be platform independent. Since the distributed network is often a heterogeneous environment consisting of different network nodes/devices made by different vendors, the administrator may have to prepare a different version of the program for different platforms. Without the platform independent characteristic, the code network mobility is less attractive. So code mobility combined with platform independence are desirable in the hot-swapping technique.

2.1.10 OS Support

The hot-swapping software needs support from the operating system. Without the support of the underlying operating system, it is impossible to conduct software hot-swapping. Traditionally, software is required to be pre-compiled and pre-linked in order to become executable. Any changes to the software must be followed by recompiling and/or rebuilding. However, the dynamic creation, dynamic deletion and dynamic linking of instances of software modules will happen during the process of on-line upgrading, so the operating system is required to support these activities.

As is mentioned in Section 2.1.4, the application services activities must be synchronized with the swap activities in order to manage the state of the application. The implementation of this kind of synchronization mechanism could be significantly simplified through support from the operating system.

Among those ten basic requirements, Code mobility and OS support are functional requirements. Without satisfying these two requirements, software hot-swapping would be impossible.

2.2 Object Oriented Paradigm as the Foundation

As hot swappable software has to be module based, one may think of the so-called structured paradigm to be the hot swapping foundation since it first proposed the modular software concept. However, it is proven that structured design/testing techniques are less successful in two respects [3]. First, the techniques were sometimes unable to cope with the increasing size of software products. The maintenance phase is the second area in which the structured paradigm has not lived up to earlier expectations. The reason for the limited success of the structured paradigm is that the structured techniques are either action oriented or data oriented, but not both.

In contrast, the object-oriented paradigm considers both data and actions to be of equal importance. A simplistic way of looking at an object is as a unified software component

that incorporates both the data and the actions that operate on the data. The object-oriented paradigm makes maintenance quicker and easier, and the chance of introducing a regression fault is greatly reduced. It is clear that the object-oriented paradigm establishes a good foundation for hot-swapping techniques.

In this paradigm, swappable software will consist of multiple objects. An object consists of both data and the actions that are performed on that data. If all the actions that are performed on the data of an object are included in that object, then the object can be a conceptually independent entity. This conceptual independence is termed encapsulation. In a well-designed object, information hiding ensures that implementation details are hidden from everything outside that object. The only allowable form of communication is the sending of a message to the object to carry out a specific action. The way that the action is carried out is entirely the responsibility of the object itself. When the object-oriented paradigm is correctly used, the resulting product consists of a number of smaller, essentially independent units. The object-oriented paradigm reduces the level of complexity of a software product and simplifies both development and maintenance. Another positive feature of the object-oriented paradigm is that it promotes reuse. Because objects are independent entities, they can be utilized in future products. The reuse of objects reduces the time and cost of both development and maintenance.

All in all, the object-oriented paradigm builds a solid foundation upon which software

hot- swapping techniques can be built.

2.3 Supports from Java Architecture

By analyzing basic requirements for the software hot-swapping technique, it is not difficult to find that Java language combined with object-oriented architecture can provide excellent support and greatly facilitate the hot swap.

2.3.1 An object-oriented paradigm

The Java architecture is an object-oriented paradigm includes the following four parts [9]:

◆ The Java Virtual Machine

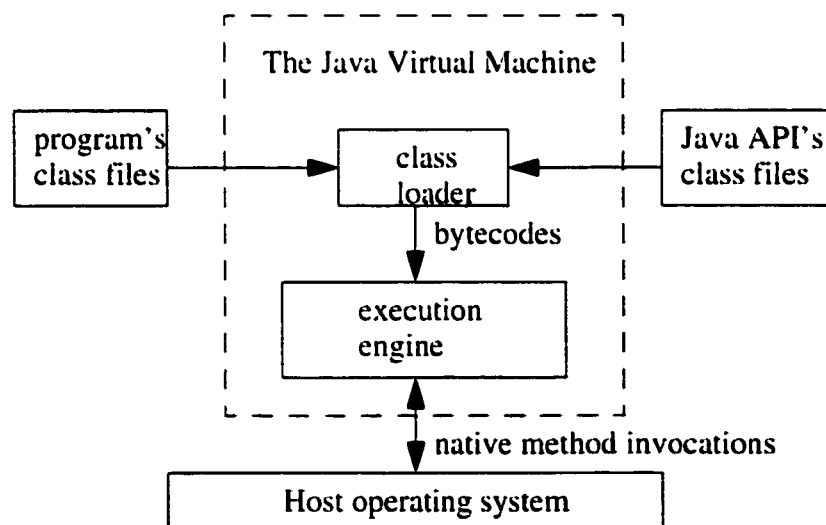


Fig. 2.3: The Java Virtual Machine

As shown in Fig. 2.3, the Java Virtual Machine contains a class loader and an execution engine. The class loader loads class files from both the program and the Java API and only those class files from the Java API that are actually needed by a running program are loaded into the virtual machine. The infrastructure of class loader contributes to many nice features of Java such as dynamic linking and dynamic extension, code mobility, security which will be explained in the following context. The bytecodes that class loader loaded are executed in an execution engine, which can vary in different implementations so that it can cooperate with different operating system through native method invocations. Thus, the structure of execution engine can support platform independence.

◆ **The Java class file**

The Java class file is a binary file of a Java program which can be run by the Java Virtual Machine. It is easy to be compact and can be transmitted over networks very quickly [9].

◆ **The Java Application Programming Interface**

The Java API is a set of runtime libraries that provide a standard way to access the system resources of a host computer. Java also provides Security API to support security implementation.

◆ **The Java programming language**

Java is an object-oriented language that promotes the reuse of code. Compared to C++, Java has some significant advantages in improving productivity of the developer. Because Java has restrictions on direct memory manipulation and provides automatic garbage collection, programmers need not worry too much about memory management. In addition, the Java API provides many standard libraries which also promote the reuse of code.

2.3.2 Dynamic linking and dynamic extension

Java's linking model allows extension of an application at runtime by customizing class loader objects. Through class loader objects, the application can load and dynamically link to classes and interfaces that were unknown or did not even exist when the application was compiled. In addition, because different class loader objects have different naming spaces, the Java architecture can allow different classes with the same name to co-exist in the same Java Virtual Machine without any conflict.

When an application is being upgraded on the fly, some new classes/objects, which may be either brand new or familiar to the application, will join the application at runtime. The Java Virtual Machine has no problem supporting this kind of dynamic linking and extension.

2.3.3 Synchronization service

Java supports multi-threading at the language level and it provides a monitor facility to synchronize the activities among threads. A monitor supports two kinds of thread synchronization: mutual exclusion and cooperation. Mutual exclusion enables multiple threads to work on shared data independently without interfering with each other. Cooperation enables threads to work together toward a common goal through wait and notify methods. A monitor can be associated with critical sections so that only a single thread is executed at a time.

When one upgrades applications at run time, there are two kinds of transaction co-existing, namely service transaction, which is initiated by user to request service from the S-application, and swap transaction, which is initiated by system administrator to conduct software hot-swapping. To preserve the integrity of the S-application, the service transaction and the swap transaction have to be synchronized. The monitor facility provided by Java can significantly simplify the implementation of those synchronization activities.

2.3.4 Platform independence

Java supports platform independence primarily through its Java Virtual Machine. No matter where a Java program runs, it needs only interact with the Java Virtual Machine and it accesses the underlying host resources through the API. Because the JVM and Java API are implemented specifically for each particular host platform, Java programs can be platform independent.

When a C++ program is compiled and linked, it generates the executable binary that is specific to a particular target hardware platform and operating system because it contains machine language specific to the target processor. In contrast, a Java program is translated into bytecodes that can run on any Java Virtual Machine.

If a system administrator wants to upgrade a distributed application in the network, some new classes have to be prepared by the administrator. Suppose these new classes depend on the administrator's platform and can not run on other network nodes, it is impossible to upgrade the application remotely by relying on those new classes. Therefore, platform independence is a necessary property for our hot-swapping technique.

In addition, Java can be implemented on a wide range of hosts with varying levels of resources, from embedded devices to mainframe computers. Taking advantage of Java's scalability, our hot-swapping technique has the potential to be applied on various applications.

2.3.5 Security

Networks represent a convenient way to transmit viruses and other forms of malicious code. An application or system can be more vulnerable while it opens the door for being upgraded on the fly. Some hostile code may invade the application by mimicking the identity of a new version through the channel that is preserved for maintenance. Thus hot-swapping technique must provide security services to protect the system.

Java's security model, which is one of the key architectural features that makes it an appropriate technology for networked environments, is focused on protecting end users from hostile programs downloaded across a network from un-trusted sources. The following elements make up Java's security model: the Java verifier, the SecurityManager, the class loader, and the language specification.

2.3.5.1 The Java Verifier

Every time a class is loaded, it must first go through a verification process. The main role of this verification process is to ensure that each bytecode in the class does not violate the specifications of the Java VM. It consists of the following four stages:

◆ **Verifying the structure of class files.**

The verifier is concerned with verifying the structure of the class file. All class files share a common structure; for example, they must always begin with what is called the magic number, whose value is 0xCAFEBABE. Following the magic number, are four bytes representing the minor and major versions of the compiler. At this stage, the verifier also checks that the constant pool is not corrupted (the constant pool is where the class file's strings and numbers are stored). In addition, the verifier makes sure that there are no added bytes at the end of the class file.

◆ **Performing system-level verifications.**

This involves verifying the validity of all references to the constant pool, and ensuring that classes are subclassed properly.

◆ **Validation bytecodes.**

This is the most significant and complex stage in the entire verification process. Validating a bytecode means checking that its type is valid, and that its arguments have the appropriate number and type. The verifier also checks that method calls are passed the correct type and number of arguments, and that each external function returns the proper type.

◆ **Performing run-time type and access checks.**

Finally, the verifier ensures that all variables are initialized correctly.

2.3.5.2 The Security Manager

In the `java.lang` package, there is a `SecurityManager` class which is used to define the security policy that specifies certain security restrictions on Java applications. The security policy's main role is to determine access rights.

Every Java application loaded into the JVM exists in its own namespace. An application's namespace defines its access boundary. This means that the application cannot access any resources beyond its namespace. Before an application can access a system resource, such as a local or networked file, the `SecurityManager` object verifies that the resource is inside the application's namespace. If it is, the `SecurityManager` object grants the access right; otherwise, it prevents it.

The `SecurityManager` class contains many methods used to check whether a particular operation is permitted. For example, the `checkRead()` and `checkWrite()` methods check whether the method caller has the right to perform a read or write operation, respectively, to a specified file. The default implementation of all of `SecurityManager`'s methods, assume that the operation is not permitted, and they prevent the operation from taking place by throwing a `SecurityException`. Many of the methods in the JDK use the `SecurityManager` before performing dangerous operations. In order to customize a security policy, one needs to subclass `SecurityManager` and overrides its check functions.

2.3.5.3 The class loader

The class loader works alongside the security manager to monitor the security of Java applications. The main roles of the class loader are as following:

- Loads class files into the Virtual Machine
- Identifies the package to which a loaded class belongs
- Locates and loads any classes referenced by the currently loaded class
- Verifies attempts by the loaded class to access classes outside its package
- Keeps track of the sources loaded classes, and makes sure that classes are loaded from valid sources

-
- Provides certain information about loaded classes to the security manager

Each class is associated with a class loader object. Before a class can be loaded into a certain package, its class loader must check which package the class belongs to. Once loaded, the class loader resolves the class, which means that it loads every other class that the class references. Resolving a class involves: verifying that the class has the right to access the classes it references, and ensuring that referenced classes are not loaded from invalid sources.

2.3.5.4 Java Language's Security Feature

There are a number of things that make Java's language specification secure, including:

- Array references are always checked at run-time
- There is no way of directly manipulating pointers
- Memory leaks are prevented by having the JVM perform automatic memory management
- Casts are not allowed to violate any casting rules

2.3.6 Network mobility

One of the fundamental reasons that make Java a useful tool for network environment is because it enables the network mobility of software. Java's architectural support for platform independence and security makes network mobility practical.

The Java class file plays a critical role in support for network mobility. Class files were designed to be compact so that they can be transmitted across networks quickly. Also, because Java programs are dynamically linked and dynamically extensible, class files can be downloaded and instantiated as needed.

Network mobility of software enables installation and upgrading to be automatic. Network-delivered software does not require discrete version numbers for its end users. These end users need not decide whether to upgrade nor take any special action to upgrade. Therefore, the job of upgrading distributed software can be significantly simplified.

2.3.7 Run time introspection

Java's reflection service provides a powerful capability of run time introspection, such as:

- ◆ Determine the class of an object.

-
- ◆ Get information about a class's modifiers, fields, methods, constructors, and superclasses.

 - ◆ Find out what constants and method declarations belong to an interface.

 - ◆ Create an instance of a class whose name is not known until runtime.

 - ◆ Get and set the value of an object's field, even if the field name is unknown to your program until runtime.

 - ◆ Invoke a method on an object, even if the method is not known until runtime.

 - ◆ Create a new array, whose size and component type are not known until runtime, and then modify the array's components.

2.3.8 Java's Drawback

Although the Java platform provides much support for software hot-swapping technique, it also includes some drawbacks.

The main disadvantage of Java is slow execution speed. For example, by interpreting bytecodes, it is 10 to 30 times slower than native execution. Just-in-time compiling bytecodes can be 7 to 10 times faster than interpreting but still not quite as fast as native execution [9].

As it is mentioned in section 2.3.1, Java has some advantages on memory management which can free Java developer from worrying about memory management. However, it is also a trade off, because a developer can no longer get entire state of the program which is not good for managing the state of an S-application.

Good news is that some improvement in the execution speed of Java programs can be expected. Sun Microsystems is currently working on a technology called "hot-spot compiling." which is claimed to yield Java programs that run as fast as natively compiled C++.

2.3.9 Alternatives

Java is by no means the only vehicle for software hot-swapping. There are some other programming languages that can support upgrading software on the fly. Actually, Smalltalk has some nice features that are suitable for upgrading software on the fly.

In Smalltalk, even a class can be treated as an object. So one can write a new class in a text file and use the operating system to compile it, instantiate it and run it at runtime. Smalltalk also allows programmers to change any function's implementation in the program at runtime. Another nice feature that Smalltalk can provide is object mutation. It allows one to simply assign the new object to the old one and they are automatically mutated. As a result, all the clients of the old object switch to the new one transparently.

However, Java is much more promising than Smalltalk, which is the main reason that we deploy this technique using Java.

2.4 Potential solutions for software hot-swapping technique

No matter how much support that existing technologies can provide, none of them have already met all the requirements for software hot-swapping. However, potential solutions may be found at different levels.

2.4.1 Solutions at Operating System Level

The advantage of developing a software hot-swapping technique at the OS level is that it directly provides a platform to support upgrading software on the fly. Moreover, an operating system can obtain the state of the system easily. Hence it is not very difficult to maintain the integrity of the system during the process of software maintenance.

However, it is quit complicated to develop a software hot-swapping technique at the operating system level. Moreover, the popularity of the software hot-swapping technique will be completely constrained by the new operating system. For example, a new operating system named Kea has been developed to provide a means through which kernel services can be reconfigured [11]. However, Kea is mainly target on upgrading kernel services.

Actually some operating system, such as Multics (Multiplexed Information and Computing Service) [12], has a quite flexible structure that can be adapted to support software hot-swapping. For example, a complete memory address of a Multics pointer includes segment number, word offset, byte offset, and a ring validation number; therefore, its memory address can be modified through manipulating the offset value of a pointer at run time.

Although Multics is almost out of usage nowadays because of its relatively slow speed, its flexible structure indicates a potential to support software upgrade on the fly. Similarly, the Java Virtual Machine also has such kind of flexible structure in updating object references for its garbage collection.

As we know, the Java Virtual Machine has to move objects on its heap for the purpose of heap defragmentation during the process of garbage collection. As it is illustrated in Fig. 2.4, one approach to implement this algorithm is to add a level of indirection to object references. That is, instead of referring directly to objects on the heap, object references refer to a table of object handles (named handle pool). When an object is moved on the heap, only the object handle is updated with the new location. Similarly, we can also upgrade an object by redirecting its object handle to a new object, that is, by manipulating the handle in the handle pool, one can redirect object reference to point to new class data in method area and new instance data on the heap.

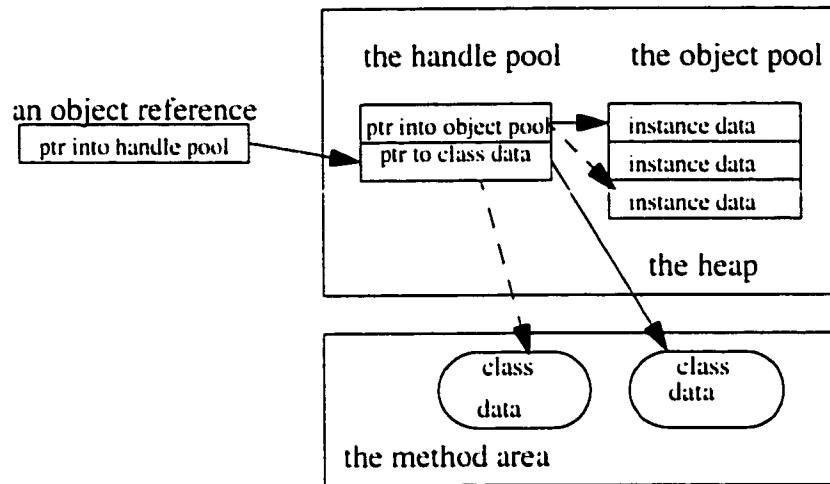


Fig. 2.4: Swap an object inside a Java Virtual Machine

However, Java does not provide an interface to let us access its object handle table and a developer could not access system state information as well. Moreover, updating object reference is just one issue of software upgrading. The tough issue is how to preserve the state of the application.

2.4.2 Solutions at system level

From the system level, a technique called dynamic system configuration [13] can be used to modify and extend a distributed system while it is in operation. In fact, it is essentially a mechanism of modifying the structure of a distributed system. This approach is essentially relying on redundant devices for system maintenance.

2.4.3 Solutions at application level

Software maintenance is an issue at the application level and many changes are specific to the application, so this problem can be solved at the application level.

The software hot-swapping technique is an approach to provide the solution for software maintenance at the application level. It should meet the ten requirements listed in section 2.1 and make it possible to upgrade the application at run time.

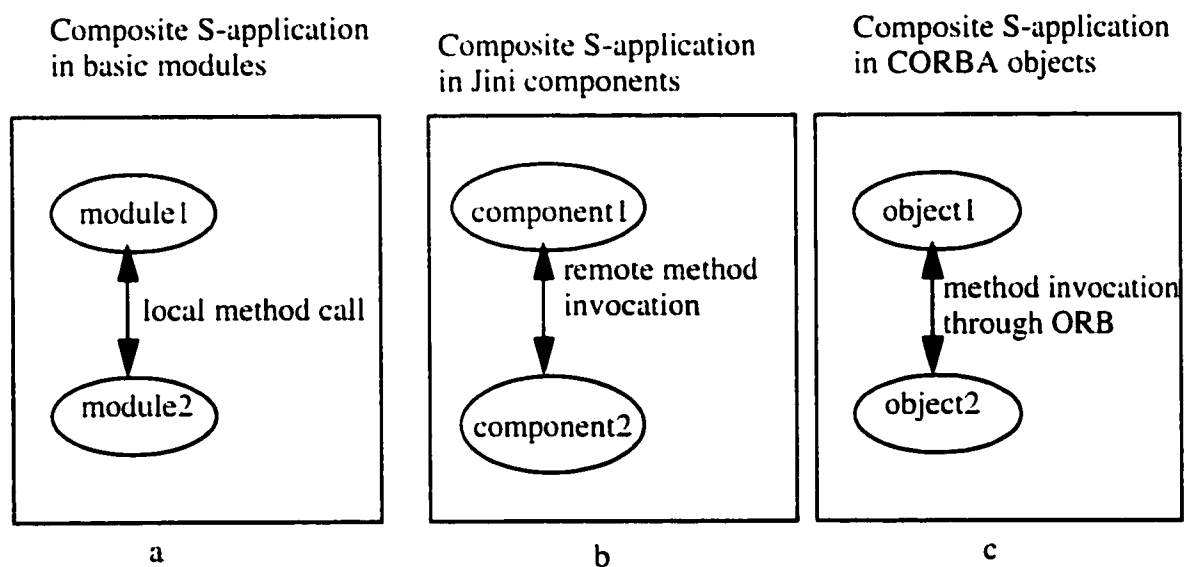


Fig. 2.5: Different proposals for composing S-application software

The infrastructure of Jini supports "plug-and-play" technique [14] and it has the potential to be used to upgrade software on the fly. A program can be regarded as a federation [15, 16, 17] of all its modules, and every module can be an add-in or subtract-out at run time. Modules can find each other through lookup services and their relationship can be very flexible. However, by comparing Fig. 2.5a and Fig. 2.5b, one can find that components in S-application have to communicate with each other through RMI instead of local method call if adopting Jini's infrastructure for S-application configuration. The performance trade off is very obvious.

A CORBA business object [18, 19, 20] is declared to be a plug-and-play component and its interoperability makes it possible to plug new applications into a shared model while preserving the integrity and security of the shared model. So this technology can be used for upgrading software at run time. However, CORBA business objects are normally complex and they communicate with each other through an ORB. From Fig. 2.5c, one can find that the performance of the application is sacrificed during the process of marshalling and un-marshalling.

2.4.4 The research assumption

Based on the preceding analysis, our software hot-swapping technique is an approach to tackling software maintenance at the application level. For performance reasons, we will not adapt Jini or CORBA business object infrastructure as our software hot-swapping infrastructure. In another words, a new software hot-swapping architecture has to be built in order to achieve the goal of maintaining software at run time.

To simplify the problem, it is assumed that the software hot-swapping architecture is built on the Java environment to take advantage of Java's features like network mobility, security, platform independence, dynamic linking and dynamic extension, multi-threading, reflection, etc.

As it was mentioned in section 2.3.9, the hot-swapping technique does not totally depend on Java technology, but rather, takes advantage of Java's support and thus simplifies our problems. In this way, we can focus on these unresolved problems and make some substantial contributions.

It is also assumed that the application to employ the hot-swapping technique is configured into some swappable modules, which are called *S-modules*, and only those swappable modules can be automatically swapped.

In the hot-swapping architecture, *a swap manager* can take control of every transaction. There is a *swap manager* for each application and all the clients access the application through its *swap manager*.

2.5 Conclusion

This Chapter defines the requirements for swappable software and its underlying principles. It is shown that in order to achieve software modularity, dynamic extensibility, code mobility, security, Java language and technology combined with object oriented paradigm establish the key foundation to realize software hot-swapping architecture which will be described in detail in the following chapters.

CHAPTER 3.0 SOFTWARE HOT-SWAPPING ARCHITECTURE

This chapter provides an in-depth description of the software hot-swapping architecture and its major components, the S-module, the Swap manager, the S-proxy, system administrator and their roles and their comprehensive services.

3.1 An overview

The environment of the software hot-swapping architecture is a typical distributed network environment. In this environment, an application can either act as a client to initiate service requests, or a server to receive service requests from its clients and provide service. Fig. 3.1 illustrates a typical client-server application. Normally a server application has to provide services for many clients, so it has to be available even when it needs to be upgraded. So in this thesis, we are going to focus on applying the hot-swapping technique on server application.

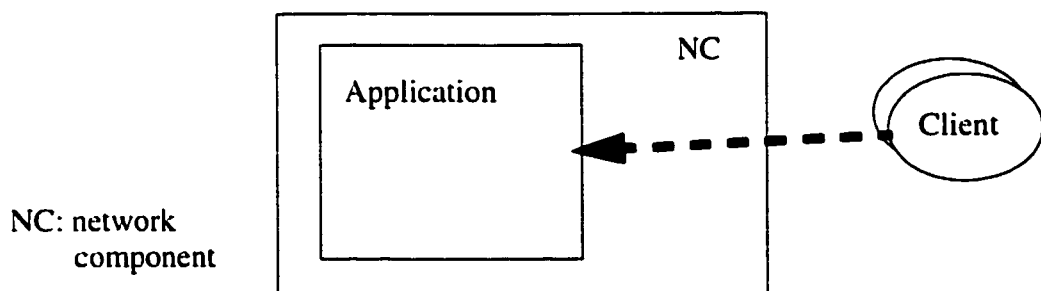


Fig. 3.1: A server application in a distributed environment

The software hot-swapping architecture consists of several fundamental elements: a specification for application configuration, a swap manager, and a system administrator.

Fig. 3.2 is the overview of this architecture.

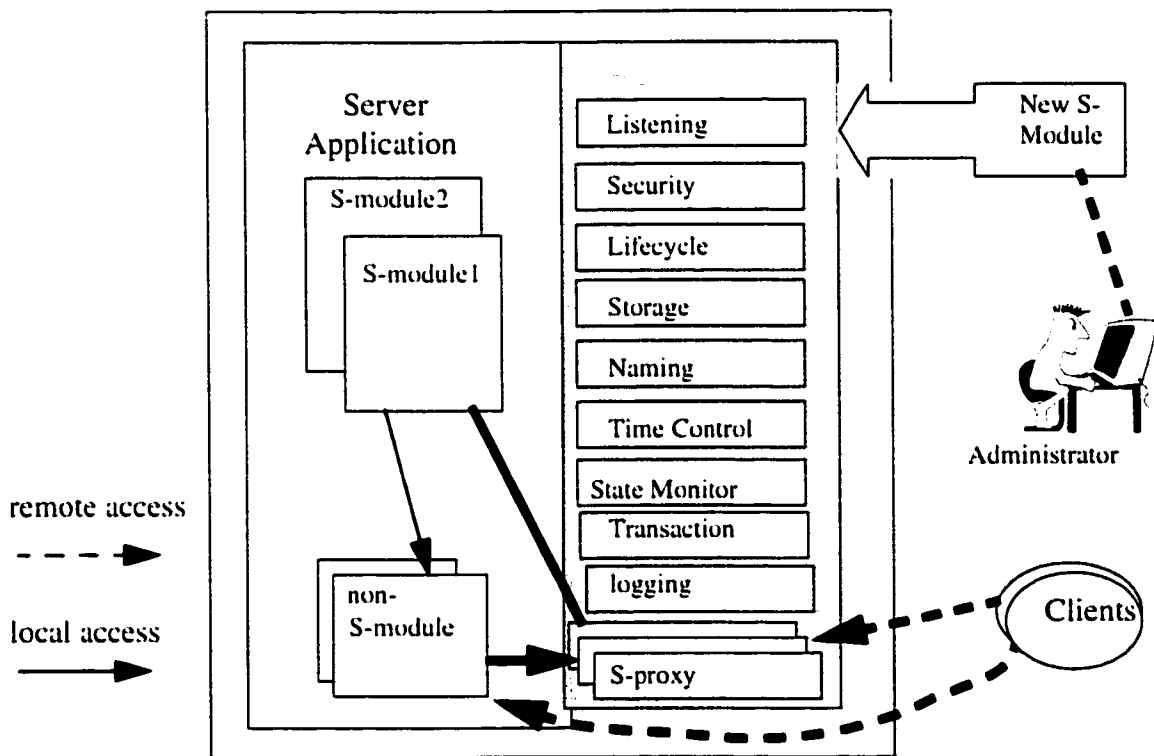


Fig. 3.2: The software hot-swapping architecture

Fig. 3.2 shows that a server application consists of multiple swappable and non-swappable modules, serving clients in network. Notice that non-Swappable module has a direct interface to those clients while Swappable modules have to go through S-proxy in order to interact with the clients.

Detailed descriptions of every component in Fig. 3.2 will be presented in the next several sections. As was discussed in last chapter, an application has to be configured into swappable modules so that it can be updated by modifying the module instead of replacing the whole application all at once. The application can be composed of multiple swappable modules (called S-modules) and non-swappable modules. Only S-modules can be swapped at runtime. In order to be able to swap an S-module on the fly, a swap manager is designed to manage its activities. Every S-module has an S-proxy to act as its inter-module communication middleman which has been discussed in section 2.1.3. The only way for the outside world to access an S-module is through its S-proxy. There is only one swap manager for an application. As it was illustrated in Fig. 3.2, the swap manager has no control of those non-S-modules. Thus from the swap manager point of view, these non-S-modules make no difference with clients of the application.

When an application needs to be upgraded, a system administrator will prepare some new S-modules and send them to the swap manager. The swap manager will instantiate these S-modules after a security check and then tries to upgrade the application automatically.

3.2 The specification of S-modules

As was discussed in the previous chapter, an application has to be configured into swappable modules so it can be replaced by unit of module instead of by unit of the whole program. A swappable module is hereby called an S-module.

An S-module, an entity that has certain attributes and behaviors, can provide certain application services at runtime, and cooperate with the Swap Manager to achieve the goal of "hot-swap".

3.2.1 The characteristics and interface of an S-module

3.2.1.1 Encapsulation and isolation

Encapsulation and isolation are key features of a S-module. With these characteristics, any software module can be implemented as an S-module. Normally a non-S-module can not be designed as an S-module mainly because it can not satisfy these characteristics.

Encapsulation means that all the information passing into or coming out of the S-module must go through its interface. This interface, hereby called the **service interface**, is to provide services for clients of the S-module. Passing information through its interface may hide a change inside an S-module from its outside world. Therefore, all the data in an S-module should be private while all the services that an S-module can provide are included in its *service interface*.

Isolation means that all the statements inside an S-module can not directly refer to other S-modules. The only way an S-module can communicate with other S-modules is through the swap manager. In another words, an S-module should not give its handle to anyone else except its swap manager. In this way, the swap manager can truly control those S-modules.

Isolation also means that an S-module can be pre-compiled and dynamically linked with the rest of the application at run time. Conversely, a non-S-module generally needs re-compilation, re-linking and restarting when it is being upgraded.

An S-module can be classified into two categories: namely, passive S-module and active S-module. A passive S-module will not initialize any operation on its outside world unless it has external stimuli; however, an active S-module may initialize operations on its outside world without any external stimuli. According to the encapsulation characteristic, a passive S-module can be controlled through its interfaces that are provided to its outside world, while an activate S-module has to implement an extra control method so that its activities that it initializes on its outside world can be controlled as well.

3.2.1.2 Identity

An S-module must have a unique identity by which the Swap Manager can identify it. However, it is not easy to achieve this unique identity in a distributed environment. An S-module is required to register with its swap manager when it is instantiated.

A new S-module may have the same name as the stale one, but their version must be different. So version is a very important index to distinguish different S-modules.

Actually version control is very important for software maintenance because swapping an S-module with a wrong version into the system may corrupt the application. An S-module

must have both a version attribute as well as an interface to access this version attribute.

```
Interface VersionControlInterface{  
    public synchronized int getVersion( );  
    public synchronized void setVersion( int newVersion );  
}
```

3.2.1.3 State

The state of an S-module summarizes its attribute values and its execution status. It is very important because it indicates the integrity of the application during the process of hot-swapping. If the swap transaction were committed, an S-module with a newer version should provide service based on the state of the stale version. If the swap transaction were aborted, the stale S-module would continue its service based on its previous state. The execution status of an S-module is of prime importance to the swap transaction; therefore, it will be further described in the next chapter.

An S-module must have the functionality to extract and retrieve its state. A new S-module may have differing attributes from the stale one, so program designers have to implement the state mapping rules at design time. However, these rules may not work perfectly at run time, the S-module should be able to handle exceptions like `stateNotMatchException`. An S-module also may not be able to provide its state all the time. For example, it is in a critical activity and can not provide its state, then an S-module of new version make catch

cannotGetStateException.

```
Interface StateControlInterface{  
  
    public StateObject getState() throw cannotGetStateException;  
  
    public void setState(StateObject o) throw stateNotMatchException;  
  
}
```

3.2.1.4 Administration interface

S-modules are managed by a swap manager that can control the activities of those S-modules and manage the transaction of swapping S-modules. Therefore, an S-module has to provide administration interface for its swap manager to control.

An S-module may not allow to be swapped in any arbitrary state, especially when it is changing the state of itself/system. Therefore, the swap manager may have to consult with the S-module before it starts the transaction of hot-swapping. On the other hand, an S-module should have the opportunity to decide whether or not it is ready to be swapped.

The swap manager also has to manage the S-module while it is swapped out, so that it can give up the occupied resources and wait for garbage collection. This kind of administration

interface may look like this:

```
Interface AdministrationInterface{  
  
    public synchronized boolean isReady( );  
  
    public void cleanup( );  
  
}
```

3.2.1.5 Persistence

Persistence is necessary to maintain the state of an S-module. Persistence along with transactional-based updates provides system recoverability. To ensure the consistency of the service application, in which the S-module is involved, the system resources held by the stale S-module, such as opened files, in-service communication channels, etc., should be able to released to or transferred to the new S-module.

It is not necessary to configure all the modules in an application into S-modules. Sometimes, it is impossible to do so because some modules in the application do not satisfy those characteristics that described in section 3.2.1.1. On the other hand, the reason why an application is configured into S-module format is to prepare for change. Therefore, the parts in the application that are likely to be changed in the future ought to be configured into S-module format. Although a programmer can not completely foresee which part of the program will be changed later, there are still some rules to follow.

3.2.2 Prepare for S-module

An S-module needs to be prepared for change. However, accommodating change is one of the most challenging aspects of good program design [10]. The goal is to isolate unstable areas so that the effect of a change will be limited to S-modules. There are three steps:

1. Identify items that seem likely to change.
2. Separate items that are likely to change. Compartmentalize each volatile component identified in step one into its own module, or into a module with other volatile components that are likely to change at the same time.
3. Isolate items that seem likely to change. Design the inter-module interfaces to be insensitive to the potential changes. Design the interfaces so that changes are limited to the inside of the module and the outside remains unaffected. Any other S-modules using the changed module should be unaware that the change has occurred. The S-module's interface should protect its secrets.

There are some areas that are likely to change, so they should be taken into consideration when a program is to be configured into S-modules.

◆ Hardware dependencies

This includes interfaces with disks, tapes, communications ports, and so on.

- ◆ **Input and output**

Input/output is relatively volatile area. If the application creates its own data files, the file format will probably change as the application becomes more sophisticated.

- ◆ **Nonstandard language features**

Using nonstandard extensions to a programming language is prone to change when the program is applied to a different environment.

- ◆ **Difficult design and implementation**

It is a good idea to hide difficult design and implementation details because they might be poorly done and require redesign. Compartmentalizing and minimizing the impact of these bad design or implementation choice might have on the rest of the system.

- ◆ **Status variables**

Status variables indicate the state of a program and tend to be changed more frequently than other data.

- ◆ **Business rules**

Business rules are the regulations, policies, and procedures encoded into a computer system. Such rules tend to be the source of frequent changes. For example, security

protocol needs to be updated periodically.

◆ **Anticipating changes**

When thinking about potential changes to a system, the effect or scope of the change should be inversely proportional to the chance that the change will occur. If a change is likely, make sure that the system can accommodate it easily.

3.2.3 Negative effects of OO

Although object-oriented technology promotes maintenance, it also has some negative effects on software maintenance. There are two obstacles that are specific to the maintenance of object-oriented software.

◆ **Inheritance**

A maintenance programmer has to study the complete inheritance hierarchy to understand the derived classes. The class hierarchy may not be laid out in a linear fashion. Instead, it is generally spread over the entire product.

The advantage of inheritance is that new leaves can be added to the inheritance tree without alerting any other class in the tree. However, if an interior node of tree is changed in any way, then this change is propagated to all its descendants. Thus, although inheritance can have a major positive influence on software development, it also has a negative impact on

software maintenance.

◆ **Polymorphism and dynamic binding**

Polymorphism and dynamic binding are very powerful aspects of object-oriented product. They decouple the caller and callee and therefore relax the type system of the programming language. However, they can also have a deleterious impact on maintenance by forcing the maintenance programmer to investigate a wide variety of possible bindings that might occur at run time, and hence determine which of a number of different methods could be invoked at the point in the code.

3.3 S-proxy

No matter how isolated an S-module is, it is a logic component in a program and must be unified within the application. In another words, every piece of code in a program is linked together according to certain logic relationships. Therefore, an S-module has to have relationships with other parts of the program and perhaps with clients of the application.

However, an S-module should not distribute its handle, otherwise it will run into referential problems. The following example explains the problem.

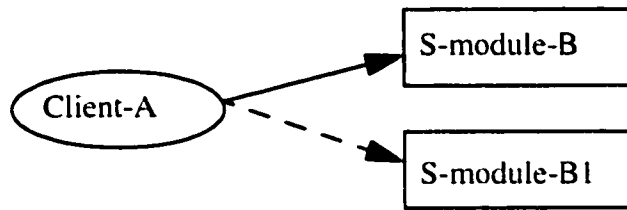


Fig. 3.3: Referential update

In Fig. 3.3, Client-A requests S-module-B for service. If we want to use S-module-B1 to replace the S-module-B, Client-A has to change the reference from S-module-B to S-module-B1. This means that Client-A must know of the change.

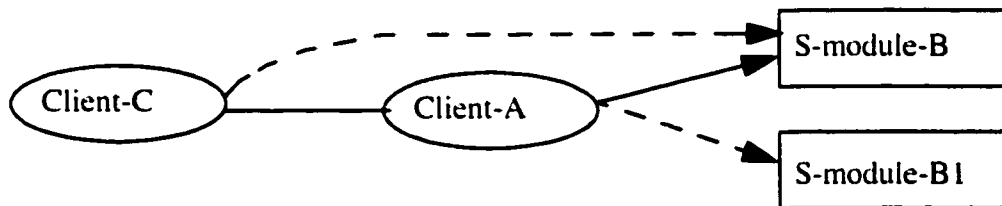


Fig. 3.4: The problem of reference propagating

However, as depicted in Fig. 3.4, if Client-A has passed S-module-B's handle to Client-C, then Client-C will retain the reference to S-module-B even if Client-A has changed the reference to S-module-B1, unless client-A informs client-B about the change. Thus, if we want to swap S-module-B with S-module-B1, we have to let all those who have S-module-B's handle switch to S-module-B1. To do so, we have the following problems:

-
- ◆ The replacement of an S-module is not transparent to its clients, as the process requires the clients' cooperation.
 - ◆ In some cases, an S-module to be swapped may have no idea about who has its handle, and thus it is difficult for the S-module to notify its clients of the change.

3.3.1 S-proxy for referential problem

An approach to solve this problem is by using an S-proxy as the delegate of the S-module. An S-proxy can hide the real handle of the S-module while those clients only get the handle of the S-proxy. When an S-module is swapped, only the S-proxy switches the handle to the new S-module while the clients remain their relationship with the S-proxy.

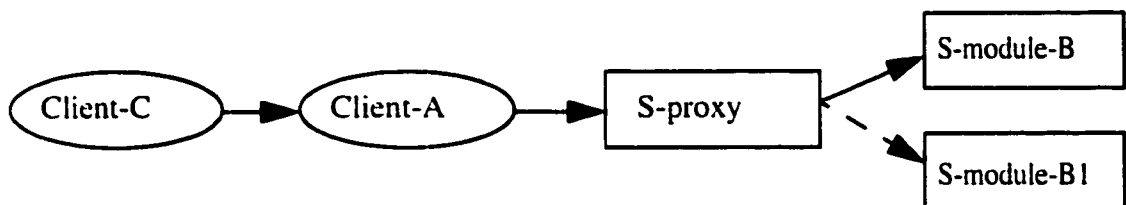


Fig. 3.5: Proxy approach for referential problem

Fig. 3.5 illustrates the proxy approach. An S-proxy is used to solve the referential problem. The S-proxy can hide the S-module-B's actual handle from its clients and deal with those clients on behalf of the S-module-B. When the S-module-B is updated, the change is

transparent to those clients. All the service requests to the S-module-B are switched to the S-module-B1 automatically through the S-proxy.

Although the proxy approach is not the only solution for the referential problem, it suits the hot-swapping architecture very well. Some other solutions are mentioned in [24], such as the observer pattern approach and the mediator pattern approach, but they are not as practical as the proxy approach.

Actually, the proxy approach is very popular in many distributed object technology such as RMI, CORBA, Jini and Voyager. However, our S-proxy does not need the process of marshalling and un-marshalling. Hence, even it has introduced some overhead into the application, it is designed to not have too many side effects on the performance of the application. A quantitative performance measurement could be done in further work.

The name S-proxy comes from the proxy pattern [23], because it has the same service interface as an S-module. However, an S-proxy not only acts as a delegate of its S-module but also plays an important role in the hot-swap transaction. Its service for a hot-swap transaction will be presented in the next chapter.

3.3.2 Functional modification and extension

The drawback of the S-proxy approach is that the service interface of an S-proxy is the same as the stale S-module that it originally represented, and the S-proxy, which is not

swappable, can not be upgraded at run time. This drawback means that a new S-module can not have an interface different from the stale one. For example, Fig. 3.6 shows that a new S-module comes into service by replacing the stale S-module. The new S-module has a newMethod interface that does not exist in the stale S-module as well as in its S-proxy. Because the outside world can only access the new-S-module through the S-proxy, then there is no way that the newMethod in the new S-module can be invoked. Obviously, this kind of limitation would make the so-called software upgrading impractical if left unresolved.

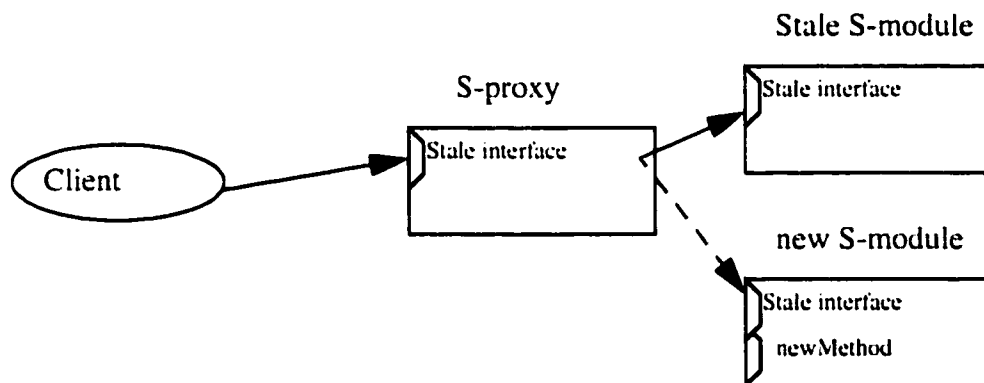
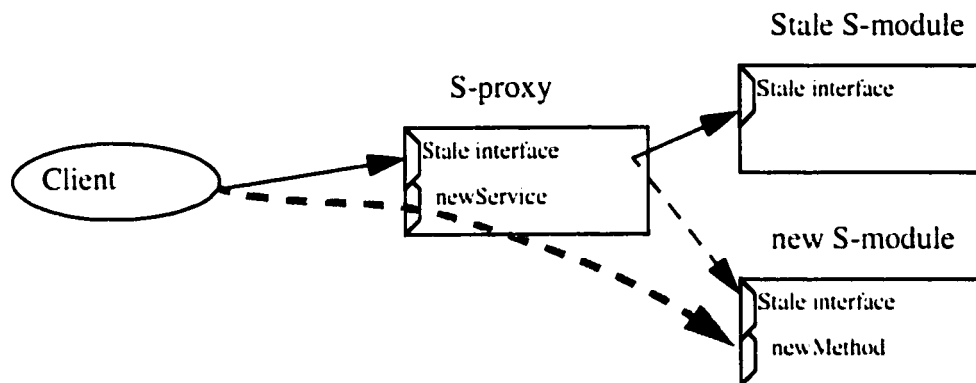


Fig. 3.6: Functional extension problem

Although an S-proxy can not foresee the new functionality of a new S-module, it can prepare for the change. By deploying Java reflection technique, an S-proxy can provide a newService interface to adapt to the case when a new S-module has a new method.

In Fig. 3.7, a new client that knows the newMethod of the new S-module can call the

newService function of the S-proxy with the parameter of the newMethod's name and parameters. Then, the S-proxy can invoke the newMethod of the new S-module.



```
Interface NewServiceInterface {  
    public Object newService(String methodName, Object[] args);  
}
```

Fig. 3.7: NewService interface for an S-proxy

Unfortunately, the above solution can only support incremental functional modification and extension. In another words, it can not support decremental functional modification. This means that a new S-module has to keep all the interfaces that its corresponding stale S-module has provided. Otherwise it will violate Java's type system. However, it is not difficult for a new S-module to continually support those interfaces that the stale S-module has already provided.

3.4 Swap manager and its services

The swap manager is the core of the hot-swapping architecture. It acts as a broker and manager among those S-modules and their clients, new S-modules and those stale ones. S-modules of an application including new and stale versions together with their S-proxies are managed by a swap manager to achieve the goal of software hot-swapping.

Although an S-proxy is tightly coupled with its corresponding S-module, from an architectural point of view, it actually belongs to the swap manager. According to the encapsulation characteristic of an S-module, the outside world must have its handle to access it. However, according to its isolation characteristic, an S-module can only give its handle to its S-proxy, so that the swap manager can take control of it. An S-proxy can coordinate this complicated relationship very well.

3.4.1 The services of swap manager

A swap manager provides the following services:

- ◆ **Listening service**

The Swap Manager has a listening service which is always listening on a specific port to receive a message sent by the system administrator.

- ◆ **Security service**

As was described in chapter 2, security is a very important issue for software hot-swapping technique. When a swap manager receives message from a system administrator, it has to make sure the following issues:

- The confidentiality of data
- Authentication of the data sender
- Integrity of the data sent
- Nonrepudiation; a sender cannot deny having sent a particular message.

Java Security API provides many ways of verifying that messages that are sent by the administrator and were not modified in transit. Normally, digital signatures and certificates can be used to ensure the security of those messages.

A digital signature is a string of bits that is computed from some data and the private key of an entity. A system administrator can generate a digital signature for its message by using the jarsigner tool or API methods, then send to the swap manager with the message and the signature together with the public key, which is corresponding to the private key used to generate the signature. Therefore, the swap manager can use the public key to verify the authenticity of the signature and the integrity of the message.

If the swap manager needs to ensure that the public key itself is authentic before reliably

using it to check the signature's authenticity, the system administrator can supply a certificate containing the public key rather than just the public key itself.

As security is a big issue, more concrete research and design are needed in future work.

◆ **Lifecycle service**

The swap manager is able to instantiate new S-modules after authentication and other security checks. It is also able to remove garbage code and free resources. Java's class loader and garbage collection mechanism can support this service.

The lifecycle service also controls the state transfer of S-proxies and S-modules in the swap transaction. Moreover, it also provides the interface to allow them to join or leave the participant list of the swap manager.

◆ **Storage service**

The swap manager is able to store the messages and new S-modules that it has instantiated.

◆ **Naming service**

A swap manager provides a naming service so that every S-proxy can be found according to its name.

◆ **State Monitor**

As was analyzed in the previous chapter, it is very important to maintain the integrity of the application. The swap manager has to monitor the state of every S-module so that it can find the checkpoint to do the hot-swap.

◆ **Time control**

The main reason why an application wants to apply the hot-swapping technique is that the system expects zero or close-to-zero down time. Therefore, the swap manager should control how long it allows the hot-swapping transaction to interrupt the application service. A system administrator can assign the time constraint through message.

◆ **Logging service**

A swap manager is able to provide information about its activities so that a system administrator is able to know the upgrade record of the application.

◆ **Transaction service**

The transaction service is the main service provided by the swap manager. It is also the most important and the most complicated part of the software hot-swapping technique. The transaction service is important because it decides whether the hot-swapping transaction succeeds or not. On the other hand, it is complicated because it has to meet all the requirements listed in chapter 2 to make the hot-swapping transaction succeed. The

transaction service has to synchronize the service transaction of the application with the hot-swapping transaction, coordinate the activities between those stale S-modules and the new ones. The swap manager is also responsible for ensuring that the swap transaction is robust and efficient as well as limited by the time constraints.

3.5 Conclusions

This chapter provided a description of the overview of the software hot-swapping architecture and its major components. The next chapter will focus on describing how the swap manager provides its transaction service, with which the functionality of the S-module, the swap manager and the S-proxy will be future described.

CHAPTER 4.0 THE SOFTWARE HOT-SWAPPING TRANSACTION

The scheme under which the swap transactions are handled during the process of hot-swapping is the key to maintaining service continuity. This chapter gives a detailed description of the transaction in our hot-swapping architecture. It starts with an overview of the concept of a transaction. Then it analyses the characteristics of the transaction in our hot-swapping architecture. Because the main responsibility of a hot swap transaction is to maintain the integrity of the application, the state of the S-module is analysed and a checkpoint for the swap transaction is suggested. Consequently, the state machine of every component in the hot-swapping architecture is described. Finally, an efficient two-phase commit hot-swapping transaction scheme is presented to ensure the system performance during transactions.

4.1 An overview of transaction concept

4.1.1 The notation and the ACID properties

A transaction can be defined as a basic unit of work or the execution of a program that performs some administrative functions through the use of one or more shared system resources, and results in a very definite, but reversible change in some part of system properties or state.

A transaction is classified by its properties atomicity, consistency, isolation, and durability (ACID) [25]. These properties characterize a transaction in a number of ways as described below.

Atomicity means that if the process of a transaction is unexpectedly interrupted by failure, all preceding operations that comprised the transaction will be rolled back.

Consistency means that the transaction will always produce the same consistent result. That is, it preserves invariance.

Isolation means that a transaction's internal transient states are invisible to other transactions and parts of the system. That is, a transaction is like a single discrete unit of work no matter how complicated the internal states of that transaction are. Normally, a transaction may appear to execute serially even though it is performed concurrently.

Durability of a transaction means that its effects are persistent, reversible (if so desired), and never lost.

4.1.2 Approaches for the state consistency of an object

There are two basic approaches to guarantee the state consistency of an object during a transaction[25]:

Logging: Persistent object values are updated in place and all changes are recorded in a

log. This approach is based on a fundamental assumption that it is possible to undo an invocation on an object. It is useful when a crash occurs part way during a transaction.

Shadowing: The update of any persistent object values are deferred. The new and stale values of each object comprising a transaction are maintained in a persistent store and a switch is made from stale to new values when the transaction is committed.

Any transaction in our hot-swapping system, which should have ACID properties, is a composite operation on S-modules invoked by clients. To keep a consistent state of those S-modules, some approaches, like logging and shadowing, need to be applied depending on the nature of transaction. However, it is not practical to require all the applications, which apply the hot-swapping technique, to keep records for every step of a transaction and all the attributes of those involving S-modules. In the next context, the features of the transaction in our hot-swap architecture will be analysed.

4.2 The transaction in the hot-swapping architecture

Generally, there are two types of transactions in our hot-swap architecture. One is designated as S-application transaction which is driven by client service requests; the other is denoted as swap transaction which is a two-phase scheme and invoked by swap manager in order to swap certain S-module.

4.2.1 The S-application transaction

The operations performed by one or multiple S-modules within a S-application in order to provide services to a client, in response to the clients request, defines the S-application transaction. A S-application transaction is initiated by the request from the client, and it ends when the service is completed.

In a distributed computing environment, it is very necessary for an S-application, an application to which the hot-swapping technique is applied, to support multiple clients concurrently. Therefore, the hot-swapping technique is designed to support concurrent S-application. It should be noted that supporting concurrency is no trivial task, as an S-module can run on multiple threads and its state transition can be very complicated.

4.2.2 The swap transaction in the hot-swap architecture

The swap transaction is a transaction invoked by a swap manager who has received a request from a system administrator to swap some stale S-modules out of the application and make new ones operational. Very likely, several S-modules in an S-application have to be updated together because they have certain relationship. To ensure the integrity of the S-application, the swap transaction can not be partially committed. In another words, it has to be a two-phase commit transaction which means either all participants can commit the transaction or none of them can do so.

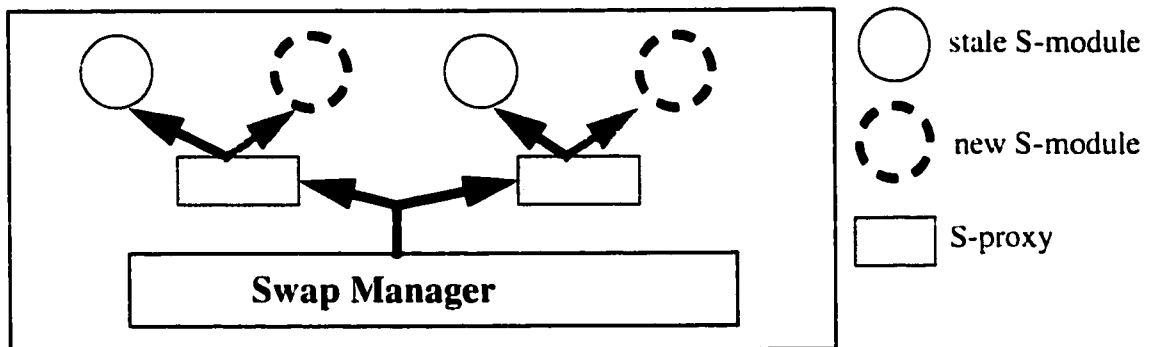


Fig. 4.1: A swap transaction

As it shows in Fig. 4.1, one unit of participants of swap transaction includes a new version of S-module, a stale version of S-module and its S-proxy. If there are multiple S-modules to be swapped, then multiple units of participants will join the swap transaction.

It must be pointed out that the swap transaction is different from transaction fault recovery. When a swap transaction starts, it does not mean that there is anything wrong with the S-application transaction. Hence the swap manager should not just abort the existing S-application transaction. On the contrary, any existing S-application transactions should be maintained.

In order to reduce the risk of crashing any S-application and to make the swap transaction robust and simple, the swap manager does not support concurrent swap transactions. Thus, the swap manager would block any new swap transaction until the swap transaction

in processing is completed.

As discussed in the previous chapter, the most difficult issue is how to keep the integrity of an S-application. The next section will target on this issue.

4.2.3 The state of an S-module and its checkpoint

Only S-modules will participate in swap transactions. The swap transaction keeps all the S-modules intended for swapping in a consistent state.

As was discussed in the previous chapter, a swap manager has to synchronize the S-application transaction and the swap transaction. If an S-module is involved in these two types of transaction simultaneously, the swap manager has no way to guarantee its consistency, because the swap transaction requires the S-module at a stable state while the S-application transaction may keep changing the state of that S-module. Therefore, it is the swap manager's responsibility to make sure that an S-module has been stopped involving into any S-application transactions before it starts to join the swap transaction.

However, according to the ACID properties of a transaction, the S-application transaction in an S-module should not stop at an arbitrary point. On the contrary, it has to stop at a swap checkpoint so that a stale S-module can reach a persistent state at which the new S-module can continue to provide service.

4.2.3.1 The state of an S-module

As was mentioned in the last chapter, the state of an S-module summarizes its attribute values and its execution status. These attributes can be classified into two categories, namely static attributes and dynamic attributes. Static attributes will not change in any S-module operation. However, a dynamic attribute may change upon some S-module operations. In another words, the dynamic attributes may change whenever the S-module is doing an S-application transaction.

The execution status of an S-module can be classified into two categories, namely busy state and idle state. The idle state means that an S-module is quiescent and is not conducting any operations. Whereas the busy state means an S-module is actually doing some operations. According to its passive characteristic, an S-module is in its busy state only when it is under external stimulus. Obviously, the dynamic attributes of an S-module may change at any time when it is in a busy state, while its attributes will not change when the S-module is idle.

For an active S-module, its busy state is more complicated than a passive S-module, because it may initialize operations on its outside world and execute operations under external stimuli at the same time. To transit from busy state to idle state, an active S-module actually has to stop initializing any operations on its outside world first and then try to complete its operations that initialized by external stimuli. Logically, it has to

follow the reverse procedure to resume an active S-module .

4.2.3.2 The checkpoint for an swap transaction

In a distributed environment, an S-application may provide service concurrently. An S-module is therefore playing different roles on different threads at the same time and its state can be very complicated. Moreover, a new S-module may have a different implementation with the stale one, so there is no way for the new S-module to find a persistent state and map it with the stale S-module which is at busy state.

Therefore, an ideal time for a new S-module to get a persistent state from the stale S-module is when the stale S-module is in its idle state and it agrees to be swapped. Then the new S-module need not worry about how to map with any change of those dynamic attributes as well as various execution states. In another words, the checkpoint for the purpose of swapping an S-module, called S-checkpoint, is at the point when the S-module is in its idle state.

The idle state also meets the requirement of synchronizing the S-application transaction and the swap transactions, that is, these two transactions do not coexist in the same S-module. However, for those S-modules which do not join the swap transaction, they can still continue with their S-application transactions and provide services. In this way, the interruption for the application is limited within the range of those S-modules who join

the swap transaction.

As was mentioned in section 3.2.1.4, an S-module must have an `AdministrationInterface` so that its swap manager can consult with it to find out whether or not the S-module agrees to be swapped. In this way, even when an S-module is at its idle state, the S-application still has its chance to decide whether or not it allows the S-module to be swapped.

To summarize, an S-module reaches its S-checkpoint must satisfy the following conditions:

- ◆ It is at its idle state.
- ◆ It agrees to be swapped.

However, whether or not an S-module agrees to be swapped is an application specific issue. In this thesis, to simplify this issue, it is assumed that an S-module is always allowed to be swapped as long as it is at its idle state, because the S-module of new version can be backwards compatible.

4.2.3.3 How to reach the S-checkpoint

Because there is a time constraint for swap transaction, a swap manager can not just passively wait for an S-module to reach its idle state. On the contrary, a swap manager

has to help the S-module to reach its S-checkpoint.

To do this, the swap manager has to block and hold all the new service requests for the S-module before it starts its swap transaction. This is essentially to synchronize the S-application transaction and the swap transaction. In this way, until the swap transaction completes, an S-module will no longer receive further invocations from its outside world except those from its swap manager.

4.2.3.4 Some Exceptional Cases

Normally, with the help of the swap manager, it will not take much time for an S-module to complete its operation and return to its idle state. Study shows that 75% of operations occur at level of milliseconds and 90% of operations occur in less than one second in a Sparc 5 workstation [10]. However, the following cases may prevent it from finishing its operations within a time constraint (e.g. the time constraint for the swap transaction):

- ◆ If one S-module depends on another S-module to complete its operation, when their swap manager tries to block and hold their new service request, their operation may run into deadlock. As a consequence, the S-module can not complete its operation and reach its S-checkpoint, so the swap transaction will be aborted because of a time-out.

- ◆ There are some operations in S-module that may exist for a long time,

such as a unbounded loop, a socket that is in communication, a running thread which seems to have no termination, and so on. In this case, the swap transaction will also be aborted due to time-out.

4.2.3.5 Avoid deadlock

As was discussed above, deadlock may occur when there is dependency between (among) two (or several) S-modules in a swap transaction. The solution for this problem is to identify every service request from those clients and distinguish them from those method invocations inside the application.

Fig. 4.2 gives an example on how to identify S-application transaction and swap transaction to avoid deadlock. Suppose the interface that the application provided to its clients like:

```
public void method1( );
```

Then inside the application, it will interpret this interface as:

```
public void method1(long txnID);
```

The extra parameter, `txnID`, is assigned by the swap manager and used as a transaction identity inside the application.

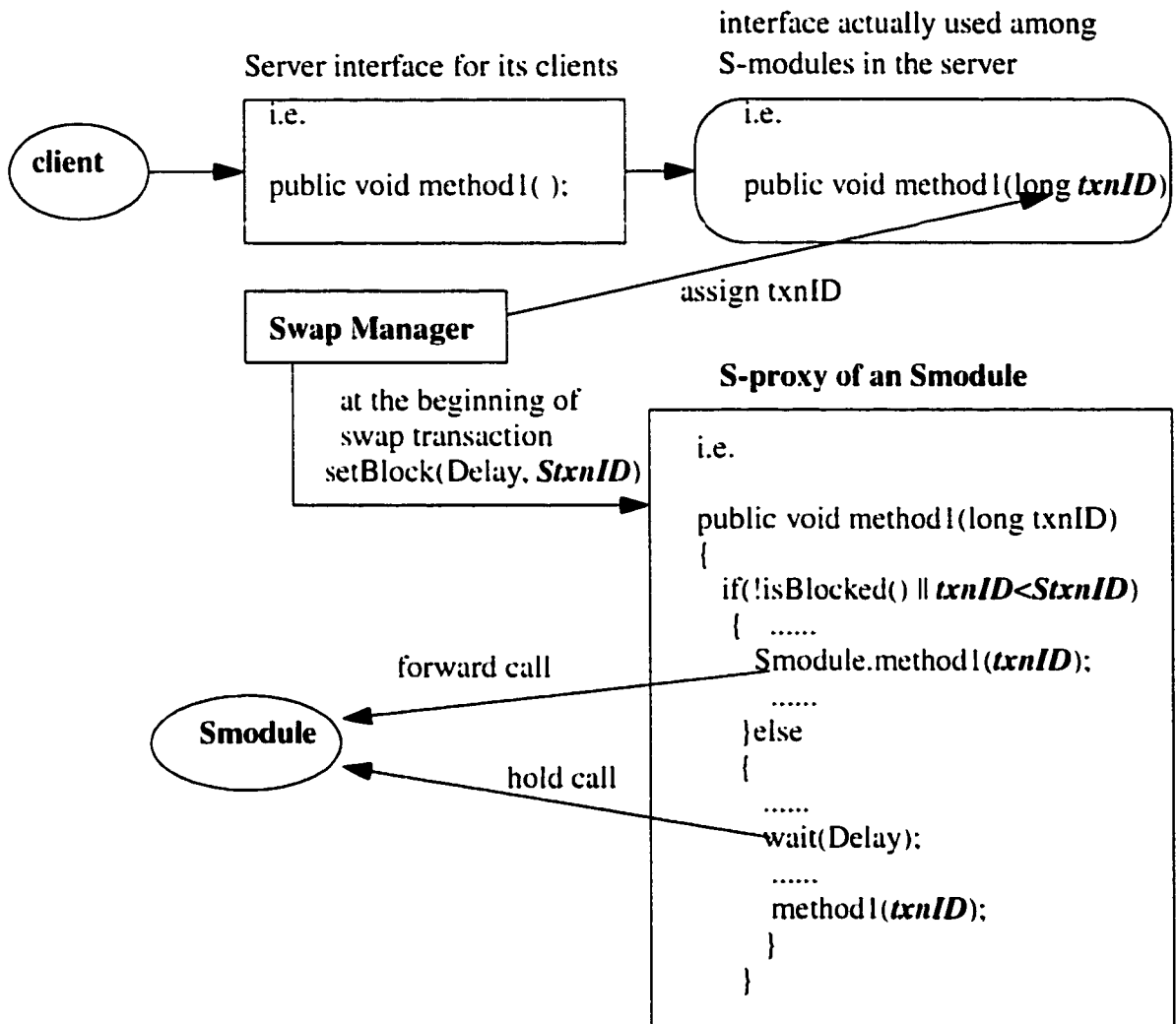


Fig. 4.2: Identify S-application Transaction and Swap Transaction

When the swap manager starts to block service requests for its S-modules, the swap transaction will also get an *StxnID*. As it depicts in Fig. 4.2, an S-proxy will only block those transactions with a *txnID* larger than the *StxnID*. In another words, it will not block

those S-application transactions started earlier than the swap transaction. In this way, every S-module has its chance to complete its stale operations and return to its idle state.

4.2.3.6 Roll back long running operations

For long running operations, the swap manager may not be able to wait for them to complete within its time constraint of the swap transaction.

Ideally, this problem could be solved if such a long run operation could be suspended, its state and tasks can be transferred to the new S-module, and the new S-module can continue the operation. However, in reality, it is quite difficult to interrupt some operations such as I/O and multi-thread operations without losing their states [27]. It is sometimes impossible to pass the break point to the new S-module with different implementation.

Now that it is impractical to wait for those long running operations to complete and it is difficult to suspend them in the middle way and transfer its state, rolling those current transactions back to their beginning states seems a better solution for the problem. In this way, both the new S-module and the stale one can find a common point to hand over.

However, rolling those long running operations back is also not easy. A scheme is needed to suspend or even interrupt the application transaction at a proper point. Moreover, if those operations have already changed the state of the application, then the roll back

transaction has to roll back the state of the application as well.

To summarize, as explained above, logging and shadowing are two proposed general strategies to keep the consistent state of transaction and can be used in this roll back strategy. However, it should be realized that there is no generic scheme to tackle the long running operation case. In other words, the mechanism for rolling back some long running operations could be very application specific and the implementation details of this strategy is out of the scope of this thesis.

In the next section, state diagrams of S-module, S-proxy and Swap Manager are presented to describe the swap transaction.

4.3 The state machine of an S-module

Fig. 4.3 shows the state machine of an S-module. An S-module starts out idle as soon as it is instantiated.

For the S-module invoked by the application, it may progress to the busy state when it responds to a stimulus. It moves back to its idle state when there is no operation in processing. As described in the preceding context, the swap manager will notify and help the S-module to move back to its idle state.

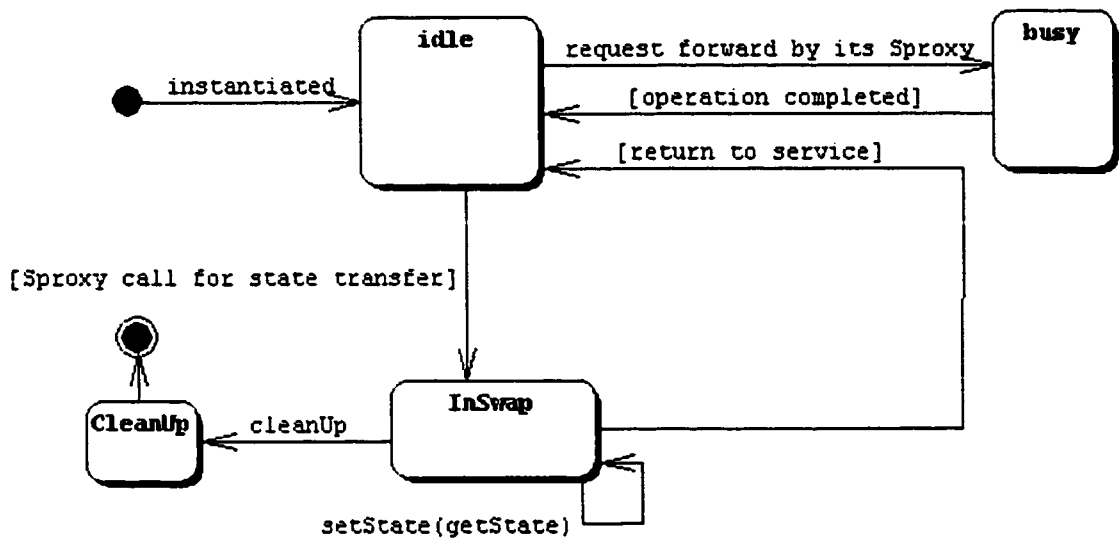


Fig. 4.3: The state machine of an S-module

For the new S-module, it stays in its idle state after being instantiated by the swap manager. It effectively enters its InSwap state upon the command from its S-proxy.

If the swap transaction succeeds, the new S-module will come into service and the stale one will move to its Cleanup state. Otherwise, the stale S-module will resume to provide service while the new one moves to its Cleanup state and is ready to be garbage collected.

Although the state transition of an S-module is apparently simple, it is critical for a swap transaction. A successful swap transaction relies on the stale S-module to progress from its busy state to the idle state as well as transfer data to the new one.

An S-module may have the following reasons to abort the swap transaction:

- ◆ The stale S-module is not ready to be swapped.
- ◆ The stale S-module can not move from the busy state to the idle state within the time constraints.
- ◆ The stale S-module throws an exception during the process of getting its state.
- ◆ The new S-module fails in the processing of mapping states that it has extracted from the stale S-module.

4.4 The state machine of an S-proxy

In the structure of the software hot-swapping architecture, an S-proxy plays an important role in delegating its S-module to deal with clients. It also has the capability to support incremental interface changes to a new S-module.

In the swap transaction, the S-proxy acts as a middle-level transaction manager in coordinating the activities between the stale S-module and the new one. Through this transparent layer, the simplicity and robustness of the swap transaction is greatly enhanced.

Fig. 4.4 shows the state machine of an S-proxy. After an S-proxy is created, it starts out

idle and moves to InService state on the invocation of the clients to provide service on behalf of its S-module. In normal operation condition, the S-proxy stays in the InService state relaying communications between client and S-modules.

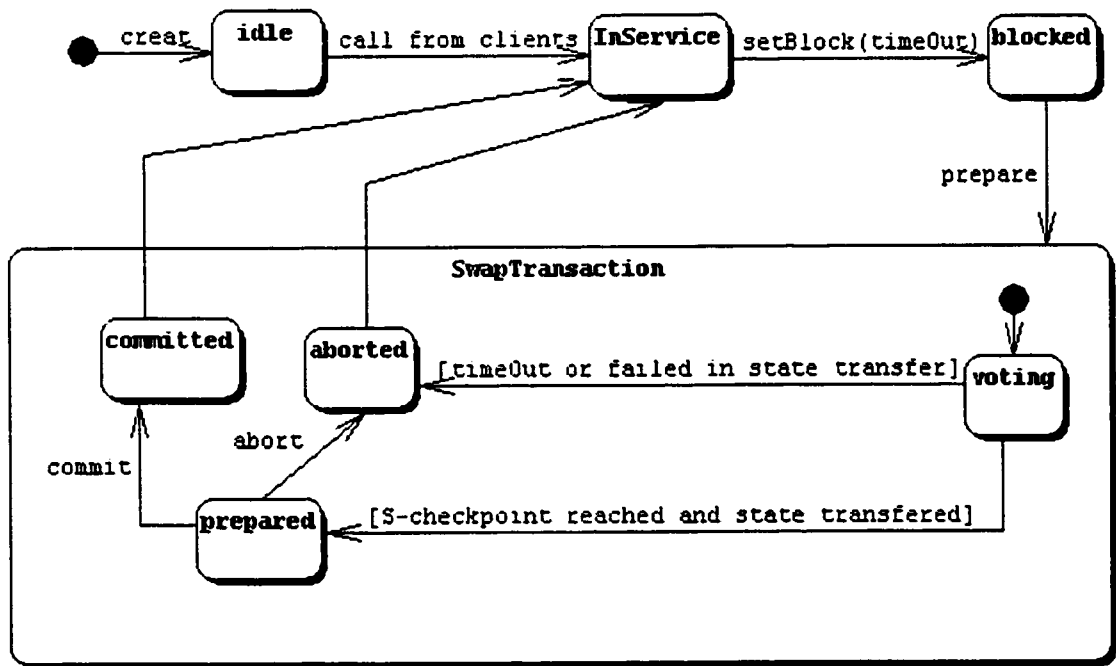


Fig. 4.4: The state machine of an S-proxy

When the Swap Manager calls an S-proxy for setBlock, the S-proxy moves to the blocked state and begins to block and hold all the new calls to the S-module.

When the Swap Manager asks the S-proxy to prepare, the S-proxy moves to its voting state at which it waits for the swapping status of its S-modules and reports the result to

the Swap Manager. There are two possible return values for voting:

- ◆ If the stale S-module fails to reach its S-checkpoint for hot-swap, or the new S-module fails to get the state from the stale one, or the Swap Manager call for abort, the S-proxy must signal this with a return of **ABORTED**, then effectively entering the aborted state.

- ◆ If S-modules (both the stale and the new one) are ready for swap, the S-proxy will return **PREPARED**, thereby move to prepared state. The S-proxy stays in the prepared state until it is told by the swap manager to commit or abort.

If the S-proxy receives a prepare call when it is not in blocked state, it throws `cannotPrepareException`.

If S-proxy receives an abort call, whether in the blocked, voting, or prepared state, it should move to the aborted state, make the stale S-module resume and ask the new S-module to cleanup.

If S-proxy receives a commit call when it is in the prepared state, it should move to the committed state. In this state, the S-proxy will replace the stale S-module's handle with the new one's, and make the new S-module operational. Meanwhile, it removes the stale S-module's handle and asks it to cleanup.

4.5 S-manager in swap transaction

4.5.1 The state machine of the Swap Manager

Fig. 4.5 shows the state machine of the Swap Manager. The Swap Manager's initialized state is the listening state, in which it waits for messages at a specific port. When it receives message from an administrator for hot-swapping, the Swap Manager moves to SecurityCheck state and does the security checking.

If the message fails to pass the security check, the Swap Manager then discards the message and returns to its listening state. Otherwise, the Swap Manager will move to the prepare state.

In its prepare state, the Swap Manager first load the byte codes and instantiate the new S-modules. If there are any problem in instantiating those new S-modules, the Swap Manager moves to its cleanup state. Otherwise, it will try to find all the participants for the swap transaction.

If Swap Manger successfully gets all the participants, it moves to its voting state, at which the Swap Manager will first ask all the participating S-proxies to intercept and hold all the new service requests for their corresponded S-modules so that they can return to their idle state. When reaching its S-checkpoint, the new S-module will extract the state from its stale S-module and try to map its state. The outcome, either **ABORTED** or **COMMITTED**, will

be reported to the swap manager.

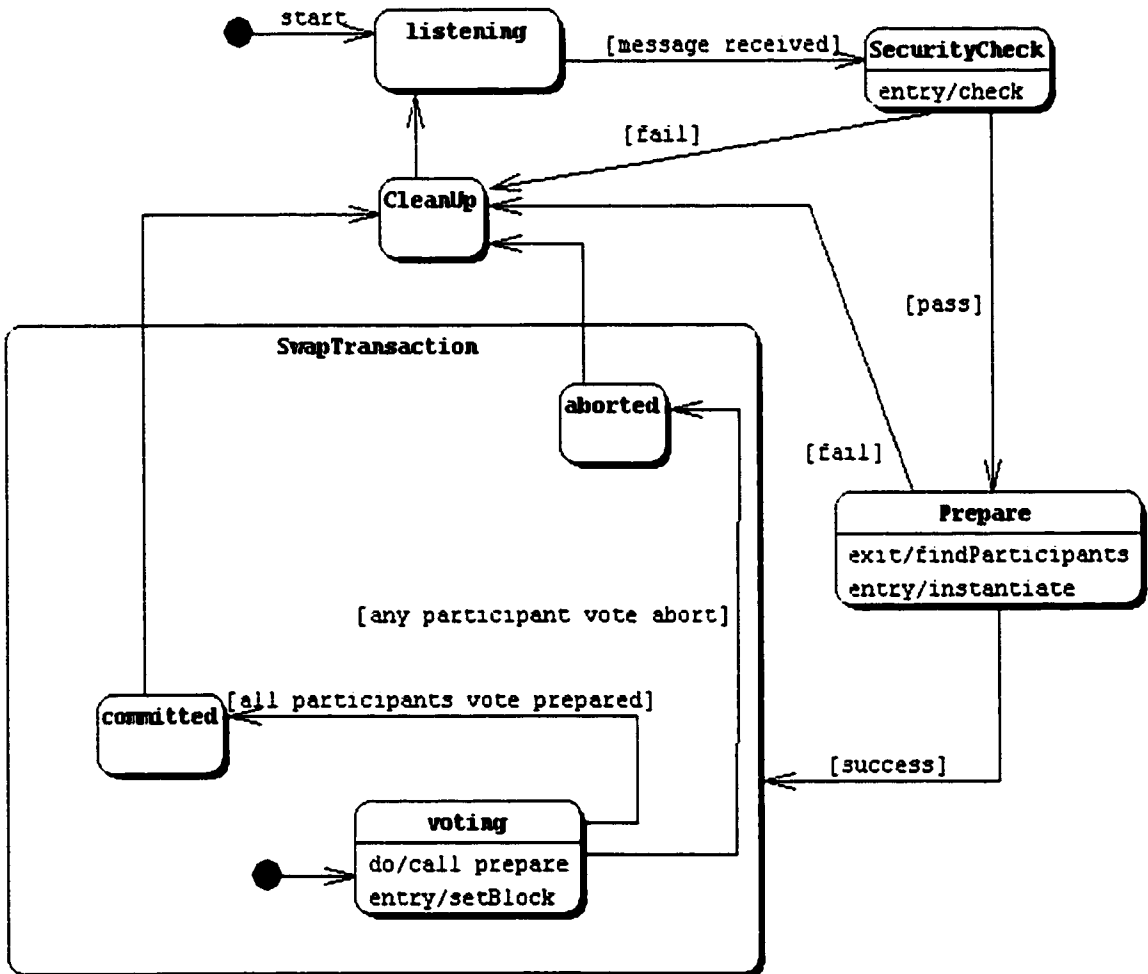


Fig. 4.5: The state machine of the Swap Manager

If any of the participants return ABORTED either because it runs into exceptions, or because it is not ready, or because it times out, the Swap Manager must abort the hot-swap

transaction. To abort the transaction, the Manager moves to the aborted state. In this state, the Manager should invoke abort on all participants even if some of them have voted PREPARED.

If all the S-proxies vote PREPARED, the Swap Manager will move to committed state and invoke the participants' commit method.

Eventually the Swap Manager will do a cleanup job and move back to its listening state.

4.6 The two-phase commit transaction model

Through the state machine of each component in the hot-swapping architecture including the S-module, S-proxy and S-manager, their roles and state transformation in the swap transaction have been described in detail. Every unit of participants in the swap transaction has three kinds of tasks to do, namely prepare-task, commit-task and abort-task.

Prepare-task is composed of the following activities:

- ◆ A stale S-module has to move from its busy state to its idle state with the help of its swap manager.

- ◆ If the stale S-module agrees to be swapped, it will transfer its state to its correspond new S-module.

◆ If the above activities are completed successfully within the time constraint, the result will be **PREPARED** and the corresponded S-proxy will move from its voting state to its prepared state. Otherwise, the result will be **ABORTED** and the S-proxy will move to its aborted state.

Commit-task is composed of the following activities:

- ◆ Every stale S-module to be swapped is removed from the application.
- ◆ Every new S-module is ready for providing the application service.
- ◆ In the meanwhile, every participating S-proxy has to move from its committed state to its InService state after receiving the commit instruction from the swap manager.

Abort-task is composed of the following activities:

- ◆ Every participating stale S-modules is ready to continue their application services.
- ◆ All the new S-modules are removed.
- ◆ Every participating S-proxy has to move from its aborted state to its InService state.

For every unit of participant, its activities in the swap transaction are sequential. That is, its prepare-task is followed by commit-task or abort-task. However, from a system point of

view, the transaction service of the swap manager can have different models to manage the activities of those participants in the swap transaction, namely a sequential transaction model and a concurrent transaction model.

A sequential transaction model means that every unit of participant in the swap transaction will line up to process its swap transaction. One participant will do its prepare-task first. If the result is PREPARED, then the next participant will do the same job. If every participant is PREPARED, then every participant will do its commit-task one by one. If anyone is ABORTED, then every participant has to do its abort-task.

A concurrent transaction model means that all units of participants process their swap transaction concurrently. That is, they will do their prepare-task simultaneously and then they will do their commit-task or abort-task according to the decision of their swap manager.

Suppose T_p denotes the time that a unit of participants need to do its prepare-task, T_c denotes the time that a unit of participants need to do its commit-task, and n denotes the number of all the units of participants that join the swap transaction. It is also assumed that T_p and T_c are same for all the units of participants. The following table shows the different costs of time for swap transaction with these two different models

:

Table 1:

The total time for swapping n S-modules	concurrent transaction model	sequential transaction model
On machine with single processor	$n*(T_p+T_c)$	$n*(T_p+T_c)$
On machine with multi-processors	T_p+T_c	$n*(T_p+T_c)$

People are prone to believe that a concurrent model always has better performance than a sequential model. Actually, on a machine with single processor, a sequential transaction model has the same performance as a concurrent transaction model.

However, on a machine with multi-processors, the concurrent transaction model is better than the sequential transaction model in terms of the performance. In this case, with the number of participants increasing, the time for the sequential swap transaction is increasing while the time for the concurrent swap transaction remains the same. As was discussed before, the swap time is very crucial for the hot-swapping technique. It not only indicates how long the swap transaction has interrupted the S-application service, but also plays an important role in determining whether the swap transaction can succeed or not. Therefore, our hot-swapping technique applies the concurrent transaction model.

The concurrent transaction model is essentially a one-thread-per-task model, because every participant will do its task on its own thread. However, creating a thread is not a low-overhead operation. In fact, many system calls are involved to spawn a thread (In Windows

NT, for example, there is a 600-machine-cycle penalty imposed every time when entering the kernel [28].) To shorten the swap transaction as much as possible, we can apply the thread pool technique to manage those transaction threads. The basic ideal is that instead of spending time to create threads during the hot-swapping transaction, we can pre-create a bunch of threads for those participants, and have them sitting around waiting for their transaction tasks. When it is time for swap transaction, simply wake up those existing threads and let them do the transaction tasks. These threads can also be recycled to perform commit-task or abort-task after prepare-task.

Fig. 4.6 shows the two-phase commit transaction model which is inspired by Jini transaction model [21]. Since many participants may join the swap transaction and the swap transaction has time constraint, to improve the efficiency of the swap transaction, we have implemented this model to make all the participants run their tasks concurrently.

At the beginning of a swap transaction, the swap manager creates a prepareJob which is a threadpool containing a finite number of prepareTask threads. Each threads runs one prepareTask of a unit of participants and all the threads run concurrently.

If all the units of participants return PREPARED, the swap manager will create a commitJob consisting of many commitTasks to invoke commit on each participant. Otherwise, the swap manager will create an abortJob and ask all the participants to abort.

If only one unit of participants joins the swap transaction, the swap manager simply invokes its `prepareAndCommit` method, and decides to commit or abort the transaction according to the result returned.

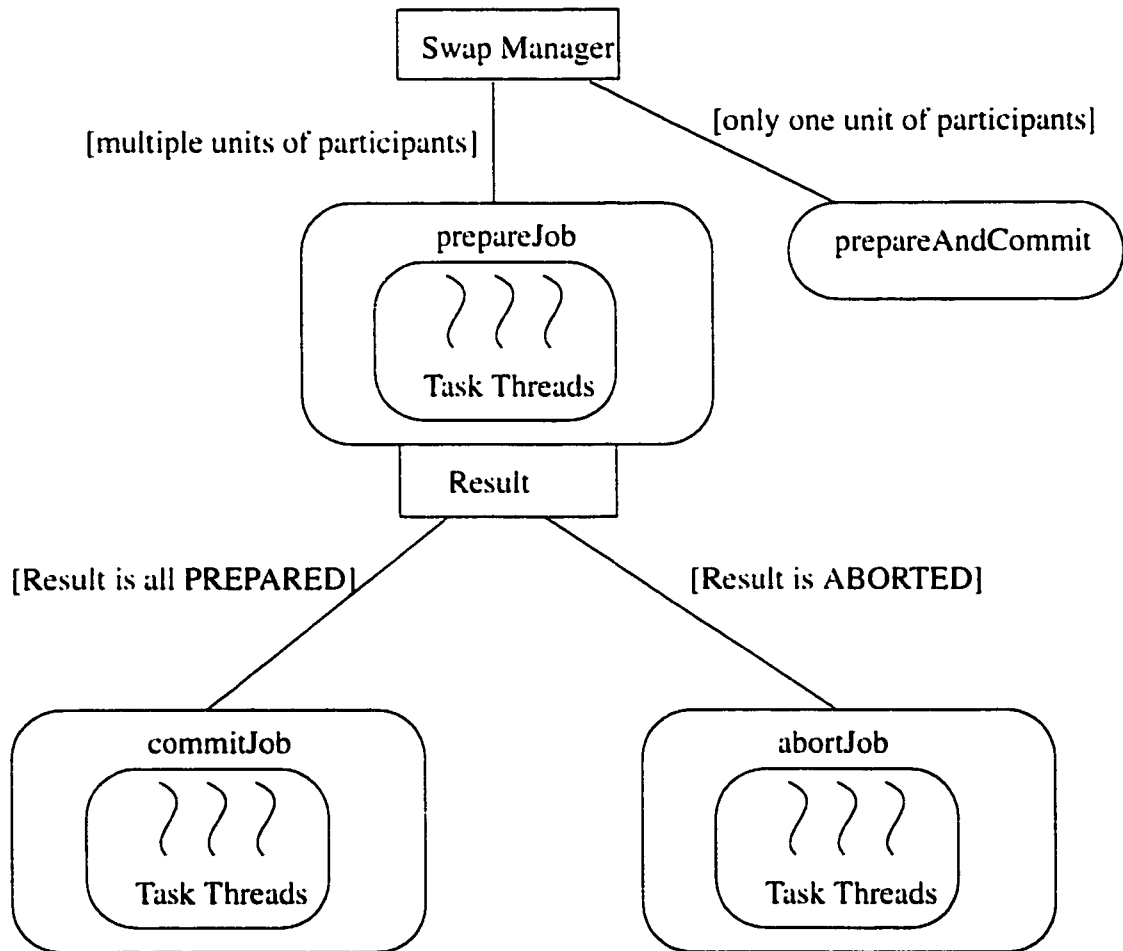


Fig. 4.6: The two-phase commit transaction model

CHAPTER 5.0 IMPLEMENTATION AND APPLICATION

In the proceeding chapters, the fundamentals, the architecture and the transaction of our software hot-swapping technique have been introduced. This chapter will focus on the implementation and application issues related to the software hot-swapping architecture and its transaction. The overall framework and implementation scheme including use case diagram and class diagrams is illustrated in detail. Hot-swapping application have been applied in various scenarios and results are presented to validate the concept of this research.

5.1 The implementation of the software hot-swapping architecture

The design of the software hot-swapping architecture follows the object-oriented design methodology and UML (Unified Modeling Language) is used to present the use case diagram, class diagram and sequence diagram.

5.1.1 Use case diagram

Fig. 5.1 shows the use case diagram of the software hot-swapping architecture. It depicts the fact that clients of the application only interact with the application service and the hot-swapping transaction is transparent to those clients. The administrator is designed to

only interact with the listening service and time-control service of the swap manager, the swap transaction is meant to be fully automatic so that the error-prone and time sensitive actions are solely handled by Transaction Service. New S-modules may interact with many services of the swap manager and may join the application service after the swap transaction commits.

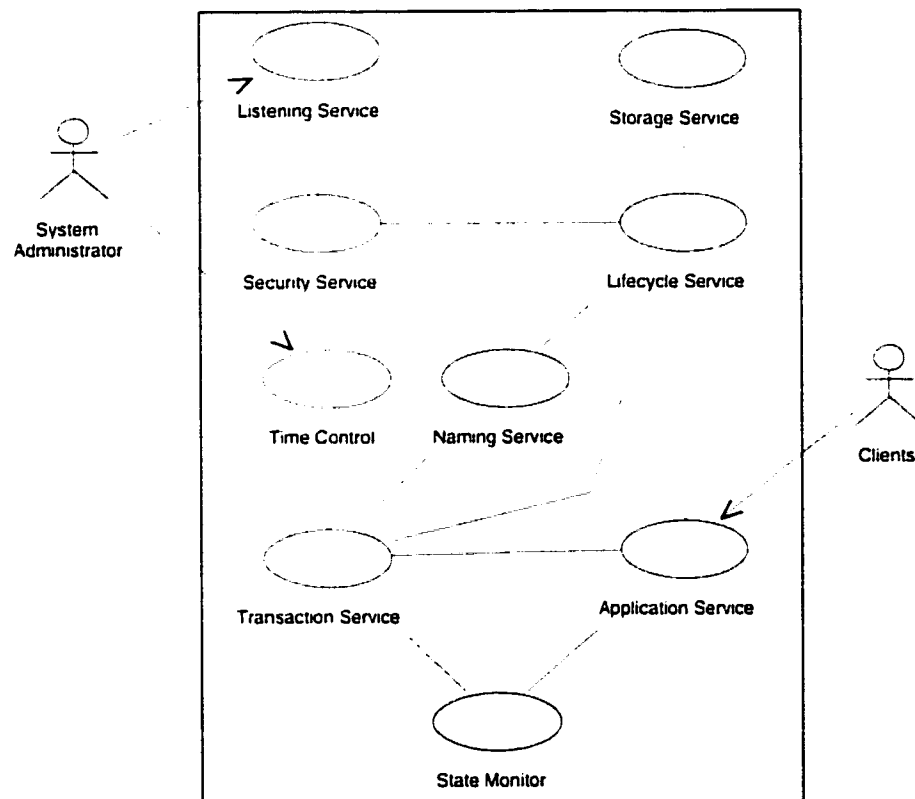


Fig. 5.1: Use case diagram for the software hot-swapping architecture

5.1.2 The class diagram

After presenting the requirements of the use case, the hot-swapping architecture is

designed. In Fig. 5.2, the class diagram shows only the pertinent classes and interfaces in the software hot-swapping architecture.

According to Chapter 3, the class diagram comprises several kinds of classes for **SwapManager** (i.e., Transaction, Security, SCreator, Naming, StateMonitor and Sproxy.) **Application** (i.e., ISmodule, Smodule), **Administrator** (i.e., SMprotocol, NewCommer) and **Client**. In the following two sections, the interface of S-proxy named IProxy and a sequential diagram will be used to describe those relationships among those classes.

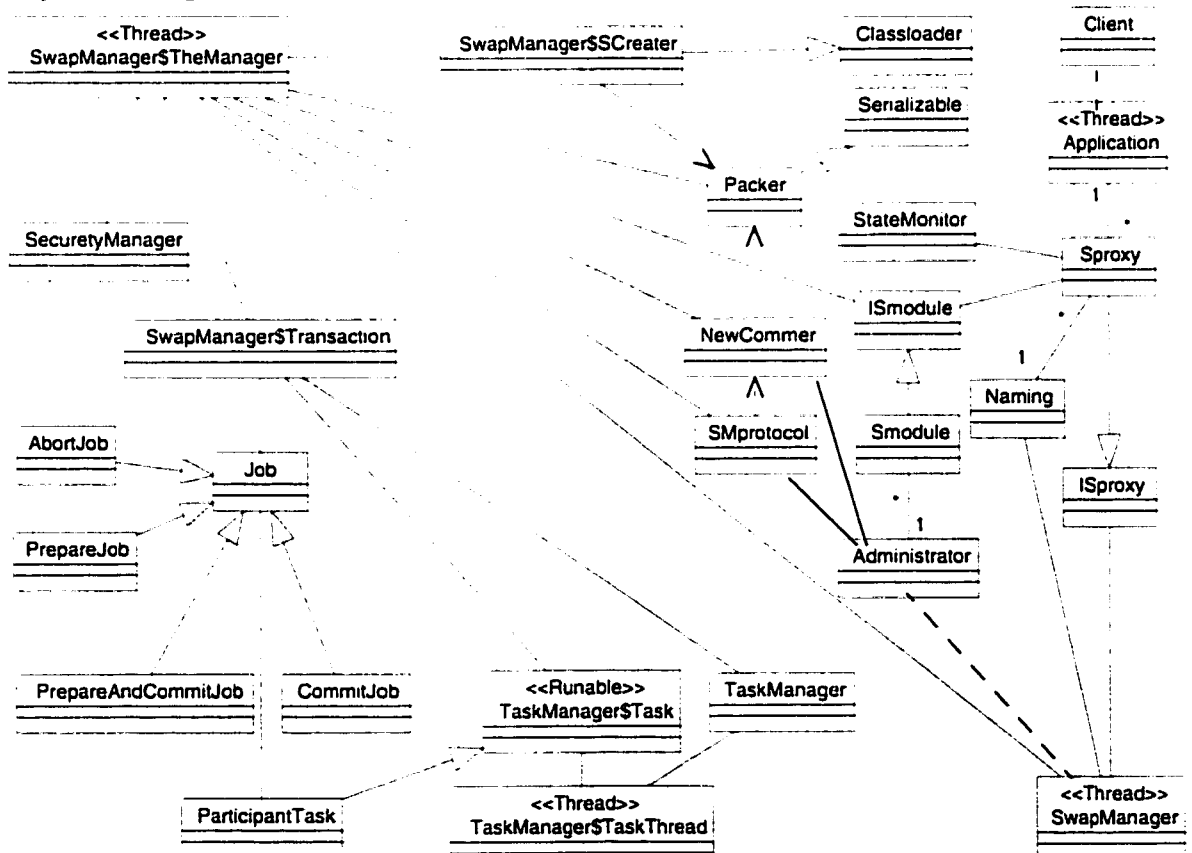


Fig. 5.2: Main Classes Diagram for the Software Hot-swapping Architecture

5.1.2 Interface for the S-proxy

The interface for the S-proxy can be classified into several categories, namely `ServiceInterface`, `NewServiceInterface`, `ControlInterface`, `LifecycleInterface` and `TransactionInterface`.

As it was described in the chapter 3, an S-proxy has the same service interface as its original S-module with which they are instantiated together. In addition, it also has a `NewService` interface through which any new methods of its new S-module are all accessible.

Through the `ControlInterface` of an S-proxy, its swap manager can block and hold service requests for the corresponding S-module, in order to help the S-module transform from its busy state to idle state. It must also be able to release the block when the S-module wants to resume its service. Every S-proxy also has an instance of `StateMonitor` which can automatically probe whether the S-module has reached its S-checkpoint or not by using an internal procedure counter.

```
Interface ControlInterface{
    public synchronized void setBlock(long Delay, long StxnID);
    public synchronized void unBlock( );
    public synchronized boolean isBlocked( );
}
```

Every S-proxy provides a LifecycleInterface to allow its new S-module to register and come into operation.

```
Interface LifecycleInterface{  
    public void register(ISModule newSModule);  
    public boolean upGrade();  
}
```

The TransactionInterface of an S-proxy is to enable the S-proxy to join the swap transaction. As we described in last chapter, the swap transaction is a two phase commit transaction.

```
Interface TransactionInterface{  
    public int prepare(long waitFor) throws UnknownTransactionException;  
    public void commit() throws UnknownTransactionException;  
    public void abort() throws UnknownTransactionException;  
    public int prepareAndCommit(long waitFor) throws UnknownTransactionException;  
}
```

Fig. 5.3 shows the relationship among clients, other S-modules, non-S-modules, swap manager, S-proxy and its corresponding S-module.

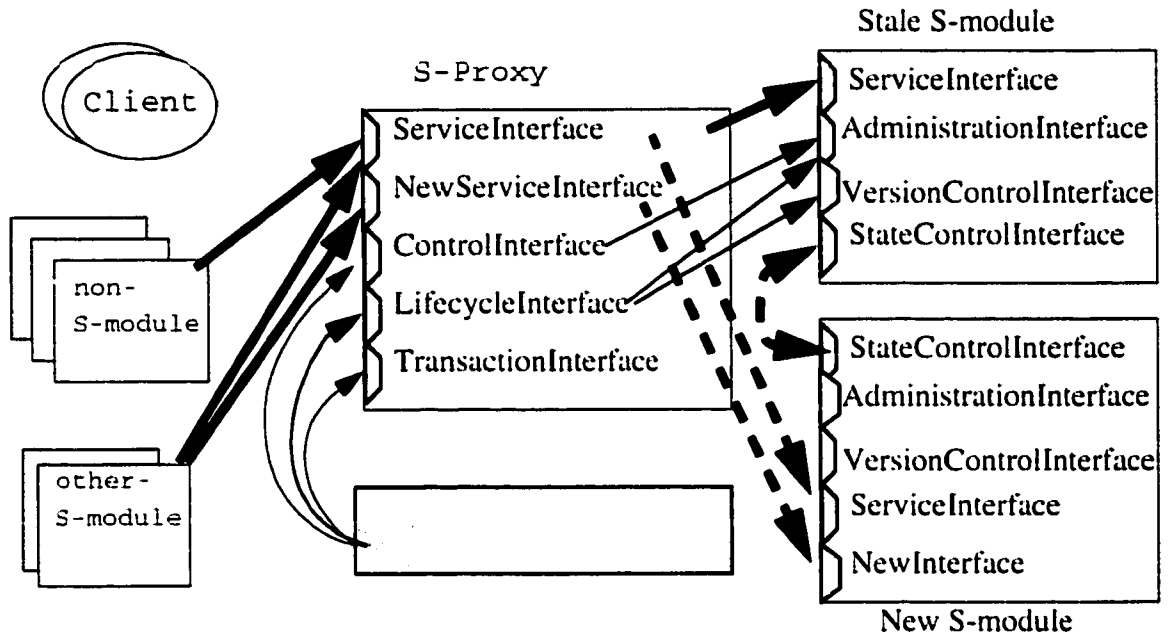


Fig. 5.3: The important roles of an S-proxy

5.1.3 Sequence diagram

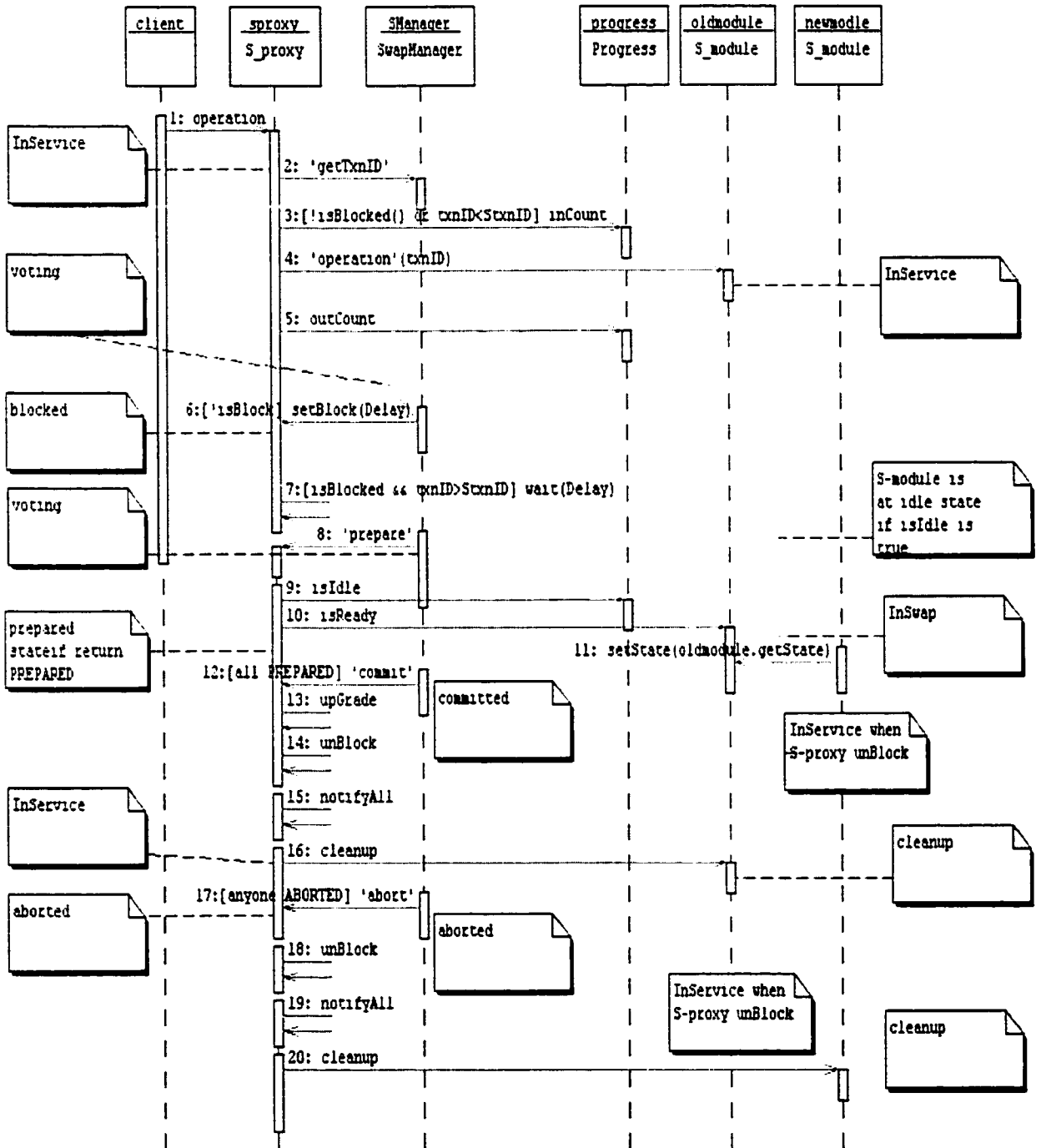


Fig. 5.4: Sequence Diagram for the case of swapping an S-module

The above sequence diagram describes the scenario of swapping an old S-module with a new S-module:

Normally, a client sends its service request to an S-proxy. If the S-proxy has not joined a swap transaction, it will forward this service request to its old S-module.

If a system administrator decides to upgrade the old S-module, then a new S-module will be prepared by the administrator and sent to the swap manager.

The swap manager is always listening at a specific port. Once it has received a message, it first does a security check. In the current design, it just verifies the password of the incoming message. However, many security algorithms can be introduced into this architecture through the Java Security API. If the message can not pass the security check, it will be discarded. Otherwise, it will be instantiated by SCreator which is a customized class loader. The security manager will also verify if the new S-module has implemented the IS-module interface.

The next step is to find the corresponding S-proxy through the naming service of the swap manager and get ready for the swap transaction. Once the swap manager starts a swap transaction, it will not get involved in another swap transaction until the swap transaction is completed.

Consequently, the swap manager will initiate a swap transaction and ask the participating S-proxy to prepare. The S-proxy will first consult with the old S-module to see if it is willing to join the swap transaction. If the old S-module is ready, then the S-proxy starts to block and hold service requests from those clients and keeps checking if the old S-module has reached its idle state (S-checkpoint). If the old S-module can not reach its S-checkpoint within the time constraints, the swap transaction will be aborted.

On the other hand, if the old S-module reaches its S-checkpoint within the time constraints, then the new S-module will try to retrieve the state from the old S-module and set up its own state according to its rule of state mapping. If there are no exceptions in this process, then the S-proxy will return the PREPARED result to the swap manager. Then the swap manager will move to the commit state. Otherwise, the swap manager will move to the abort state.

In its commit state, the swap manager lets the S-proxy to do commitJob. Then the S-proxy will switch the reference from the old S-module to the new one, release the block and let the new S-module provide service. At the meantime, the old S-module will do its cleanUpJob and be ready for garbage collection.

If the swap manager is in its abort state, then the S-proxy will do its abortJob. It will simply release service requests to the old S-module and thus the old S-module resumes its application services. In the meanwhile, the new S-module is discarded.

5.2 Application of software hot-swapping architecture

This technique has been already applied on a real application to upgrade SNMP version 3 security module [24], which demonstrated that our software hot-swapping technique has great potential for upgrading software without taking down its service. Fig 5.5 shows the structure of a swappable SNMP agent.

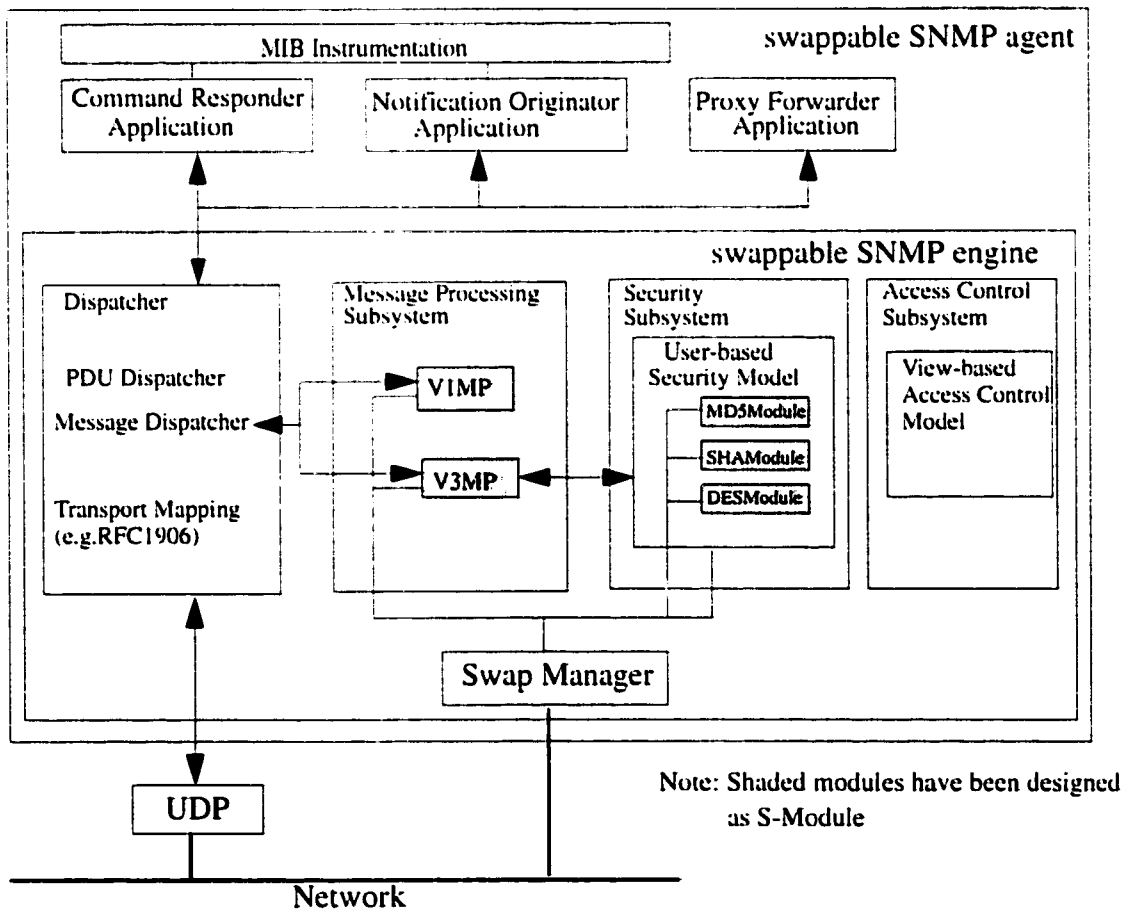


Fig. 5.5: The structure of a swappable SNMP Agent

However, since there is no state issue for security module in the SNMP V3, the aboved application can not demonstrate those most important services of our software hot-swapping architecture. On the other hand, keeping the application in a consistent state and doing hot-swapping transaction are key issues of our hot-swapping technique. Without having proved that it can handle those challenge issues properly, the hot-swapping technique would have no value. Therefore, an experimental application was designed to verify the correctness and reliability of this technique.

5.2.1 An experimental application

The experimental application was designed to use all the services that our hot-swapping architecture can provide in order to verify our hot-swapping technique and concept. The application is ideal because it was not designed to use in the industry directly. However, it demonstrated all the features of our hot-swapping architecture which the real industry application may want to use.

5.2.1.1 The objectives and basic plan for the ideal application

The objectives and basic plan for the experimental application are as follows and they are organized in such a logic fashion that complexity is increased gradually.

- (1) To support hot-swapping one S-module in a swap transaction.

This is to validate the basic utilities of the swap manager including listening service, security service, lifecycle service, naming service and transaction service. In terms of the transaction services, this application verifies the prepareAndCommit utility which can swap one S-module at a time.

(2) To support hot-swapping multiple S-modules in one single swap transaction.

Based on the last scenario, this is mainly to verify the transaction model with swapping multiple modules. This scenario will mainly cover the facility of prepareJob and commitJob.

(3) To accommodate time constraints and service request from clients during a hot swap transaction.

In the preceding application cases, there were no service requests from those clients during swap transaction and therefore all S-modules were actually at their idle state. However, in reality, an S-module has to provide service for its clients and it is normally not in its idle state.

Hence this application was to prove that the swap manager could help the S-module move from its busy state to its idle state, and automatically detect the state change of the S-module. In this process, the control facility and State Monitor facility would be verified.

If the S-module can reach its S-checkpoint within the time constraints, the swap transaction should commit. Otherwise, the swap transaction should be aborted. Therefore, if the time constraint is big enough, the swap transaction has a good chance to commit. Otherwise, it is likely to be aborted.

(4) To illustrate how the swap manager keeps the integrity of the application

This scenario was designed to prove that the facilities and the mechanism of hot-swapping architecture can keep the consistency of the application, which means that the new S-module is able to get its state from the stale one and the transactions initiated by client will not get lost.

(5) To swap several S-modules with dependent relationship

This scenario was designed to prove that the swap manager is able to swap S-modules that have dependent relationship without running into deadlock.

(6) To invoke new methods on a new S-module through its S-proxy

This scenario was designed to verify the new-service facility of an S-proxy and ensure that it can be used to access those new methods of a new S-module.

(7) To accommodate the security of the architecture

This scenario was designed to illustrate the security facility in the software hot-swapping architecture. If the password in a swap messaging could not match with the swap manager's password or the new S-module has not implemented ISmodule interface, the swap messaging will be discarded.

(8) To check the interruption of the application service

As was discussed in chapter 2, the hot-swapping technique is delay sensitive. So the down time is a critical issue for the software hot-swapping technique, and the interruption should be minimized as much as possible.

(9) To support the concurrency of the S-application during the process of swap transaction

To minimize the interruption of the S-application, the swap manager only interrupts the service of those S-modules which are participating in the swap transaction. Those S-application transactions, which those swapping S-modules are not involved in, can still be executed concurrently with the swap transaction.

5.2.1.2 The Application Illustration

To achieve the above objectives, an experimental application was designed to have all the scenarios described in the previous section. Fig. 5.6 illustrates this model which has a typical client-server relationship in a network.

The server application, which was composed by several S-modules, namely S1, S2, S3 and S4, would provide application service for its clients. The swap manager, which was registered with S-proxies including SP1, SP2, SP3 and SP4 corresponding to those S-modules, was designed to run on another thread to provide hot-swapping services for this server application. A system administrator has prepared several new S-modules, namely S11, S21, S31 and S41, to replace those running S-modules.

Through the network, the system administrator can remotely deliver those new S-modules to the swap manager. Upon completion of system security check, the swap manager is able to instantiate those new S-modules through its class loader and start the hot-swap transaction. Figure 5.6 illustrates that through S-proxy SP1, the newly loaded S-module S11 is to replace existing S-module S1.

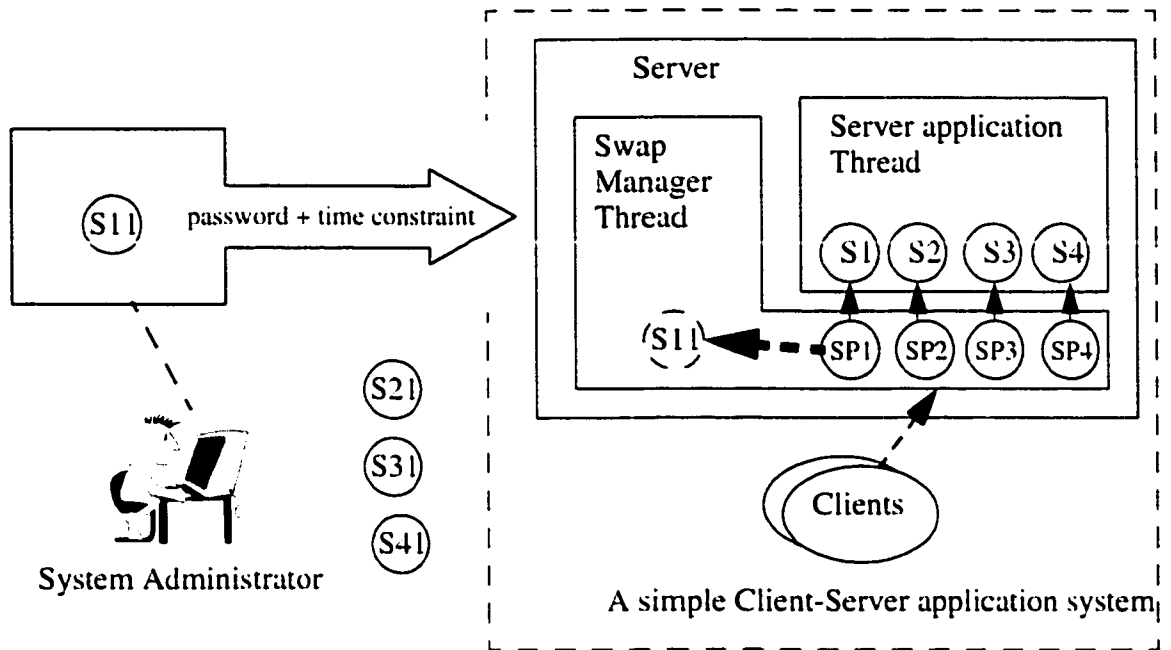


Fig. 5.6: An experimental application of our hot-swapping technique

The swap transaction can be committed based on the following two criteria:

- All the participants can reach their idle state within the time constraints.
- All the new S-modules can get state from the old ones.

To accommodate the state transfer, every S-module has static state and dynamic state.

The static state is the handle of the output frame, while the dynamic state is a counter that indicates how many times those clients have invoked the S-module.

To add the time constraint, the system administrator gives a maximum swap time indicating how long it can be tolerant for the swap transaction.

To accommodate that every S-module would take time to complete its operation, a delay is inserted into every method of those S-modules.

```
Method1( ){  
    try{  
        Thread.sleep(delayTime);  
    }catch(InterruptedException iex){ }  
  
    Output;  
}
```

If the time constraint is longer than the method execution time, then the swap has a good chance to commit. Otherwise, the swap transaction may be aborted. Therefore, the value of the delay time and time constraint can be adjusted to verify the swap transaction mechanism.

To illustrate the support of complex dependent relationships among S-modules, Figure 5.7 shows that there is a recursive call among S2, S3 and S4. It does matter that in whatever order the modules are swapped.

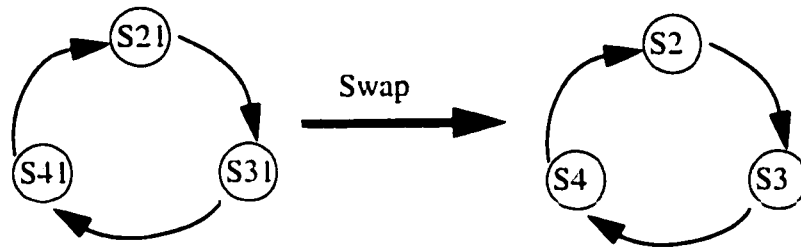


Fig. 5.7: Swap S-modules with dependent relationship

5.2.1.3 The Application panel and results

There are three panels in this application representing three main components in the hot-swapping architecture:

A server node which has application thread and a swap manager thread running.

A client node which can send many service requests to the server.

A system administrator who can prepare new S-modules and send hot-swapping request to the swap manager.

Through the administrator panel in Fig. 5.8, a system administrator can prepare some new S-modules which are listed in the NewSomoduleList box. The time constraint for every swap request can be set in Timeout box. For example, the time constraint in Timeout box is 40 milliseconds as showed in Fig. 5.8. For security reason, the panel user needs a password to be able to send out swap message.

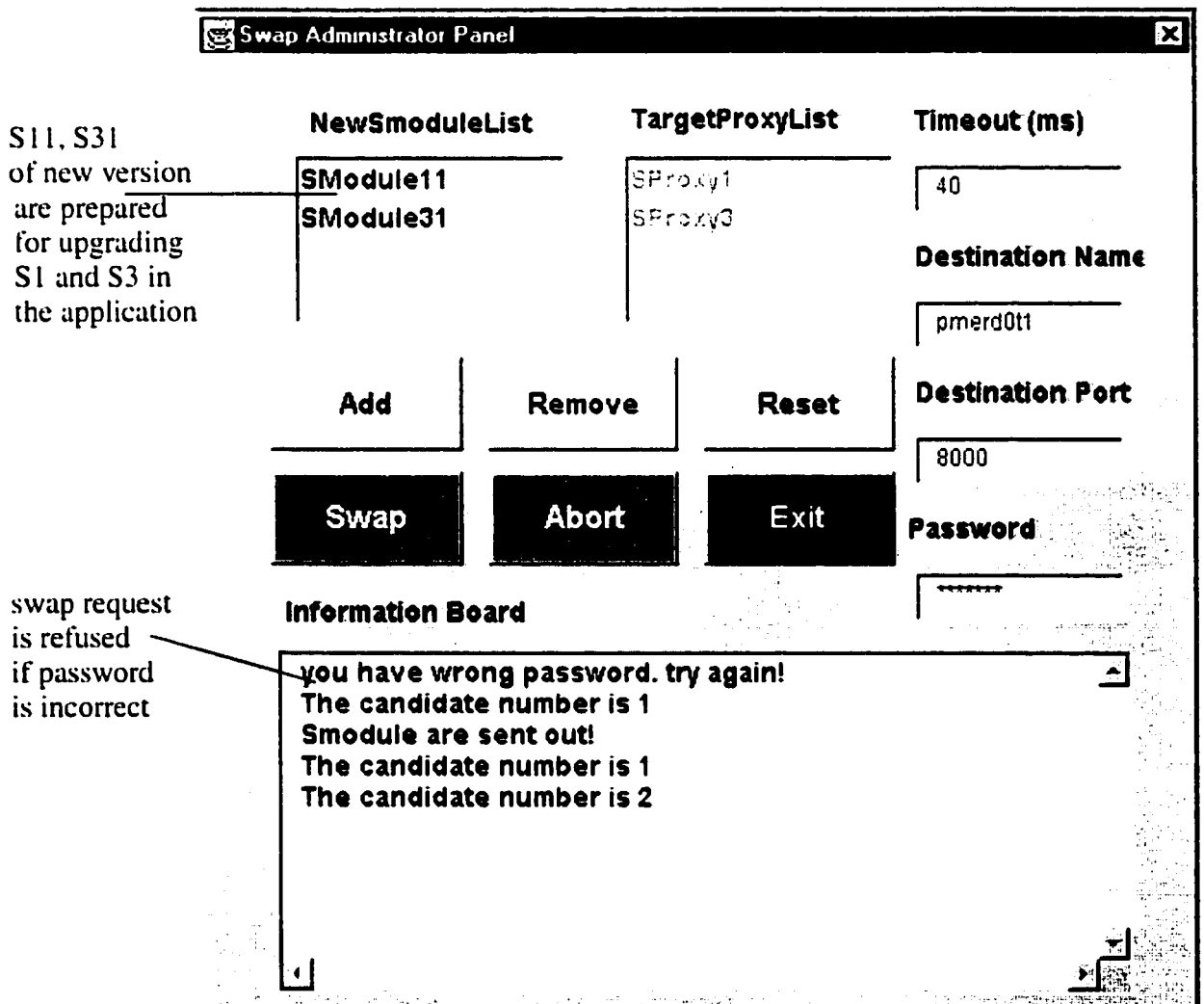


Fig. 5.8: Administrator panel

Through the client panel in Fig. 5.9, the user can indicate which S-module will be the service provider and how many service requests will be sent out to it. The number of service request can be used to verify if there is any transaction being lost in the process of hot-swapping.

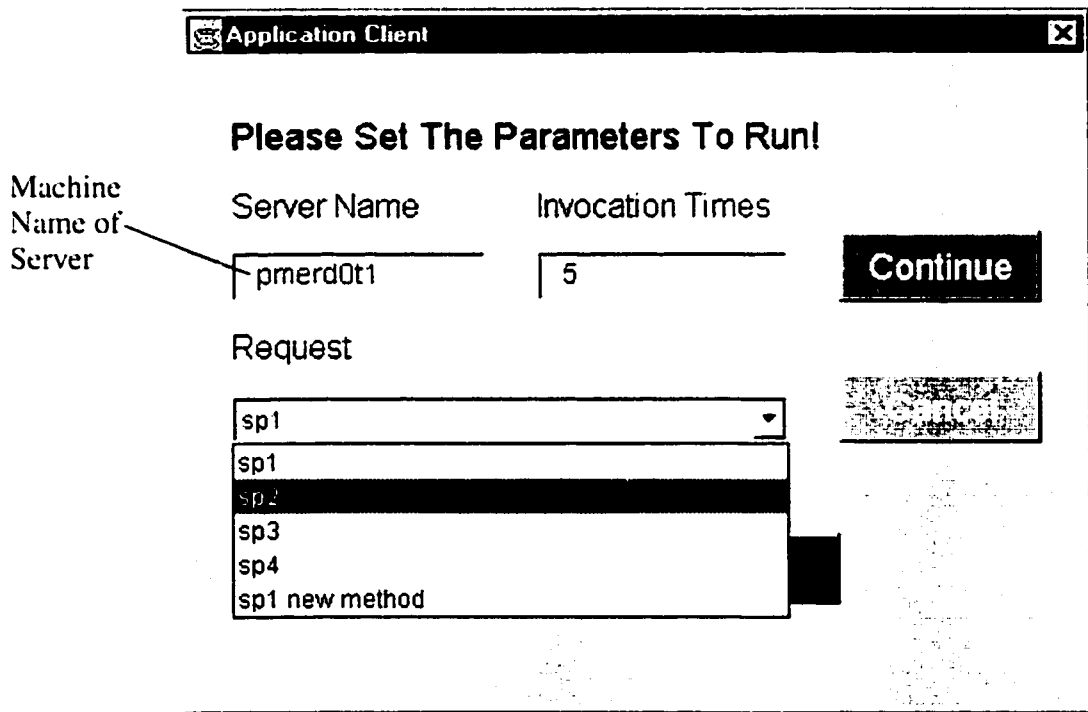


Fig. 5.9: Application Client panel

On the server panel of Fig. 5.10, there are three text boxes illustrating different outputs.

The top one is the output of S-module one which indicates the version of the S-module and how many times it has been invoked.

The middle box indicates the new method of a new S-module has been accessed through the new-service interface of its S-proxy.

The big box is the output of recursive call among S1, S2 and S3. The result indicates that the swap manager can handle the dependent relationship among S-modules without running into deadlock.

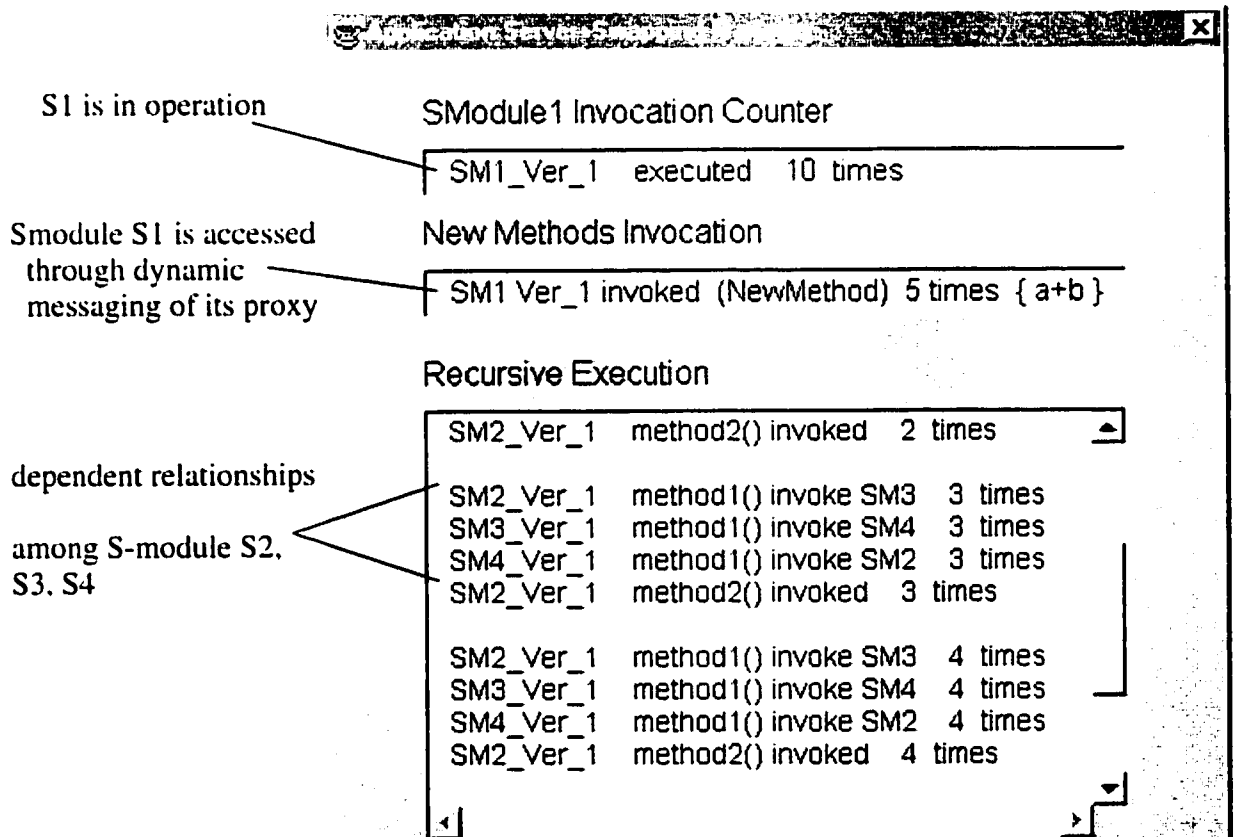


Fig. 5.10: Application server and its swap manager

Fig. 5.11 indicates that S-module S1 and S-module S3 are swapped successfully without losing their states and transactions, as the values of those counters are correct.

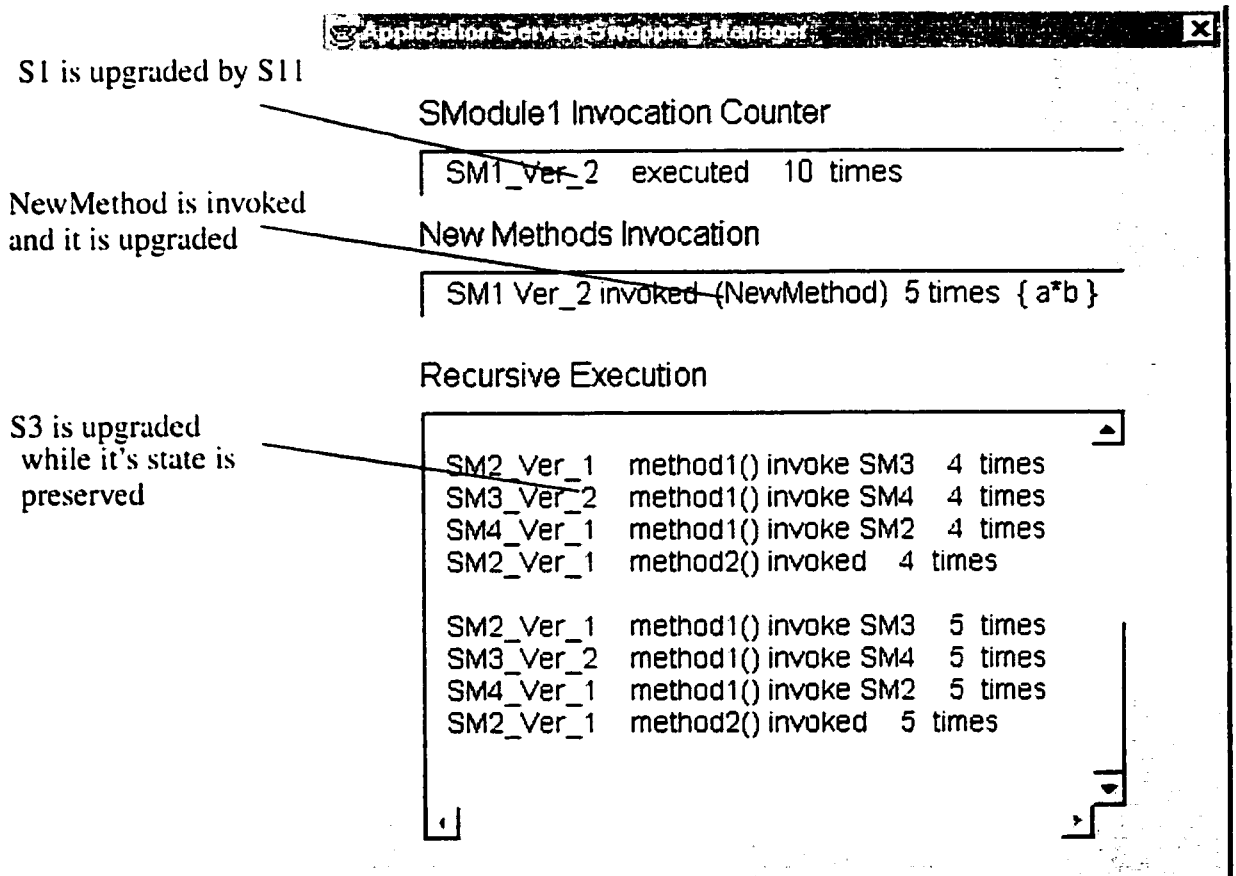
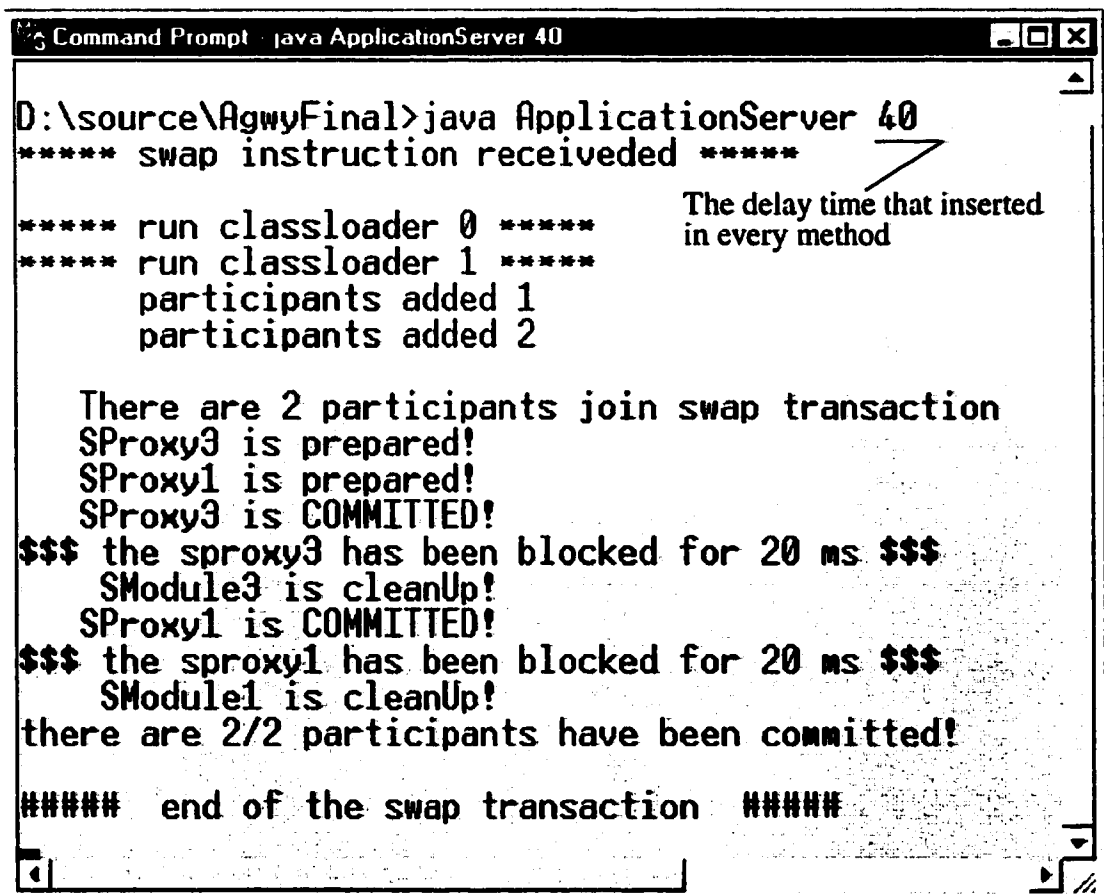


Fig. 5.11: The result of swapping S-module S1 and S-module S3

Our test proved that if the S-module is not in the idle state, the service interruption for hot-swapping transaction depends on how long an S-module can move from its busy state to idle state.

Fig. 5.12 demonstrates the procedure of swapping S-module S1 and S-module S3 in the same swap transaction. As was mentioned in section 5.2.2.2, to simulate the time of executing every operation, we hereby insert 40 milliseconds delay into every method. In this case, the service of S-module S1 and S-module S3 were blocked for a while (20 ms). Since the time constraint that given is longer than the blocking time, so the swap transaction can be commit.



```
Command Prompt java ApplicationServer 40
D:\source\AgwyFinal>java ApplicationServer 40
***** swap instruction received *****
***** run classloader 0 *****
***** run classloader 1 *****
        participants added 1
        participants added 2

There are 2 participants join swap transaction
SProxy3 is prepared!
SProxy1 is prepared!
SProxy3 is COMMITTED!
$$$ the sproxy3 has been blocked for 20 ms $$$
    SModule3 is cleanUp!
SProxy1 is COMMITTED!
$$$ the sproxy1 has been blocked for 20 ms $$$
    SModule1 is cleanUp!
there are 2/2 participants have been committed!

##### end of the swap transaction #####
```

Fig. 5.12: The print out of swapping S-module S1 and S-module S3

If the insert delay is 100 ms instead of 40 ms, then the swap transaction would be aborted because of time out. Fig. 5.13 demonstrates this scenario.

```

D:\source\AgwyFinal>java ApplicationServer 100
*** server application receive request ***
$$$$ the request is sp1
from SModule1 at state 1
$$$$ the request is sp1
from SModule1 at state 2
*** server application receive request ***
$$$$ the request is sp3
$$$$ the request is sp1
from SModule1 at state 3
$$$$ the request is sp3
***** swap instruction received *****

$$$$ the request is sp1
from SModule1 at state 4
***** run classloader 0 *****
***** run classloader 1 *****
participants added 1
participants added 2

start
  ──────────▶
timeout
  ──────────▶ There are 2 participants join swap transaction
                SProxy3 is prepared!
                swap time out!
                $$$ the sproxy3 has been blocked for 30 ms $$$
                SModule31 is cleanUp!
                SProxy3 is ABORTED!
                $$$ the sproxy1 has been blocked for 20 ms $$$
                SModule11 is cleanUp!
                SProxy1 is ABORTED!
                there are 2/2 participants have been aborted!

aborted
  ──────────▶
                ##### end of the swap transaction #####
                $$$$ the request is sp3
  
```

Fig. 5.13: The result of aborting a swap transaction

CHAPTER 6.0 CONCLUSIONS

6.1 Summary

The thesis began with a discussion of the importance of a sound software maintenance approach for distributed, mission critical software applications. Pros and cons of many existing software upgrade strategies, including patching, redundant device, parallel processor, dynamic object technology and mobile code technology, were reviewed. It was noted that none of these approaches provides an adequate yet generic solution in upgrading software on the fly without disrupting its services. It was further realized that without a sound underlying software infrastructure to provision for future evolving changes at initial design stage, a dynamic software upgrade on the fly is just not going to be realistic.

A new approach called software hot-swapping technique was therefore proposed to accomplish a simple, flexible and robust infrastructure to achieve software hot swapping. A thorough discussion of fundamental swappable software requirements such as modularity, dynamic extensibility, code mobility and network security led to the conclusion that object oriented paradigm combined Java technology provides the best foundation for the proposed hot swapping technique, which is a solution at the application level.

The next step was to proceed to introduce the overall software hot swapping architecture and its major components, which included S-module, S-manager, S-proxy and system administrator. A detailed description of the roles, characteristics and services of each component were presented, followed by an overview of the challenging issues regarding the hot swapping transactions. A transaction strategy was then proposed to guarantee the integrity and continuity of the system operations.

In order to demonstrate the applicability of the hot swapping technique, an application was developed and typical test cases were implemented and analyzed.

It is also important to realize that there are some trade-off in applying this software hot-swapping technique.

- ◆ To configure the application into S-module format, it introduces some complexities.
- ◆ To generate an S-proxy for an S-module, it occupies extra memory.
- ◆ To represent an S-module, an S-proxy introduces some overhead in processing messages.
- ◆ Although the interruption for the application service is minimized, it still takes time for those S-modules, which are to be swapped, to reach their S-checkpoint.

6.2 Conclusions

The main contributions of this thesis are

- ❶ A new software maintenance approach, the software hot swapping technique, is developed for upgrading distributed, mission critical applications. This technique provides a practical mechanism to systematically conduct hot-swap of software on the fly without taking down its services.
- ❷ Swappable module concept is introduced based on object oriented paradigm and Java technology. The proxy pattern in software hot-swapping architecture is applied and enhanced.
- ❸ A generic strategy on how to keep the state integrity of an application was defined. A mechanism for automatically detecting the checkpoint in order to engage the new S-module and disengage the stale one is also implemented.
- ❹ A two-phase commit transaction model combined with thread-pool technique was developed to ensure the robustness of the system and minimize the interruption of the software application services, inspired by the Jini transaction design.
- ❺ The challenging issues regarding the implementation of the hot-swapping technique including referential transparency problem, dependant relationship with deadlock problem, and interface changing problem was analyzed and many comprehensive solutions were provided.

6.3 Directions for Future Work

A good research topic never comes to an end. The software hot-swapping technique brings some profound concept for the software maintenance and network reconfiguration management. Based on our current achievement, future research can be performed in the following areas:

- ◆ It is very important to apply this technique to a large-scale mission critical application. By doing so, some potential problems could be identified and their solutions could be further studied as well. Thus the software hot-swapping technique will become more mature.
- ◆ Further researches are needed on how to modularize an application into S-module format and how to create a Factory facility to automatically generate S-proxy and S-module. Java bean technology could be a good reference for this direction.
- ◆ Some solutions are needed to handle long life operations, especially on how to interrupt the operation and rollback the application transaction.
- ◆ Our two-phase commit transaction model can be extended to distributed two-phase commit transaction model, which can be used to swap S-modules both on client side and server side in the same transaction. By doing this, it will allow us to have more flexibility and apply our hot-swapping technique to more applications.

◆ Strengthen the security in our hot-swap architecture. Otherwise the hot-swapping technique can bring unaffordable risk for the sensitive network applications.

◆ It is also necessary to conduct performance study on the hot-swap technique to obtain further understanding on how much the hot-swapping technique, if applied, can affect the performance and robustness of those applications.

◆ CORBA 3.0 enriches CORBA's feature with pass by value capability and makes it possible to distribute object through ORB. As CORBA is a language independent architecture, it can become a powerful vehicle for our software hot-swapping technique and extend the hot-swapping technique to that software written in languages other than Java.

REFERENCE

- [1] C. Bathe, "Patch," URL: <http://www.whatis.com/>
- [2] B. P. Lientz, E. B. Swanson and G. E. Tompkins. "Characteristics of Application Software Maintenance." *Communications of the ACM* 21 (June 1978), pp. 466-471.
- [3] S. R. Schach. "Classical and object-oriented software engineering," 3rd ed. The McGraw-Hill Companies, Inc., 1996
- [4] P. Robertson. "Integrating legacy systems with modern corporate applications." *Communications of the ACM*, Vol. 40, No. 5, Pages 39-46, 1997.
- [5] R. E. Phillips, "Dynamic objects for engineering automation" *Communications of the ACM*, Vol. 40, No. 5, Pages 59-65, 1997.
- [6] R. Laddaga, and J. Veitch, "Dynamic object technology" *Communications of the ACM*, Vol. 40, No. 5, Pages 36-38, 1997.
- [7] Y. Wang, "Integration of Mobile Agent Environment with Legacy SNMP", Thesis submitted for the Master of Engineering degree, Carleton University, August 1998.

[8] D. L. Parnas, "On the Criteria to Be Used in Decomposing Systems into Modiles."

Communications of the ACM 5, no. 12 (December 1972): 1053-58.

[9] B. Venners, "Inside the Java virtual machine." The McGraw-Hill Companies, Inc., 1998

[10] S. McConnell, "Code Complete: a practical handbook of software construction."

MicroSoft Press, 1993

[11] A.C. Veitch and N.C. Hutchinson, "Kea - A Dynamically Extensible and Configurable

Operating System Kernel", Proceedings of the 1996 Third International Conference on

Configurable Distributed Systems (ICCDs '96), 1996

[12] P. Green, "Multics Virtual Memory - Tutorial and Reflections," available at URL: [ftp://](ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html)

[ftp.stratus.com/pub/vos/multics/pg/mvm.html](ftp://ftp.stratus.com/pub/vos/multics/pg/mvm.html)

[13] J. Kramer and J. Magee, "Dynamic Configuration for Distributed Systems." IEEE

Transactions on software engineering, VOL. SE-11, No. 4, April 1985.

[14] S. K. Raza, "A Plug-and-Play Approach with Distributed Computing Alternatives for

Network Configuration Management", Thesis for Master degree, Department of Systems

and Computer Engineering, Carleton University, 1999

[15] "Jini Technology Architectural Overview" available at URL: <http://java.sun.com/products/jini/whitepapers/architectureoverview.pdf>

[16] "What is Jini" available at URL: <http://java.sun.com/products/jini/whitepapers/whatisjini.pdf>

[17] "Jini TechWhy Jini Technology Now?" available at URL: <http://java.sun.com/products/jini/whitepapers/whyjiniinow.pdf>

[18] R. Orfali, and D. Harkey. "Client/Server Programming with JAVA and CORBA", published by John Wiley & Sons, Inc, 1997.

[19] "Business Objects Interoperability Specification" available at URL: <http://www.dataaccess.com/dat/Download/Interop.PDF>

[20] "Common Business Object Facility Proposal" available at URL: <http://www.dataaccess.com/dat/Download/Boa10.PDF>

[21] "Jini Transaction Specification" available at URL: <http://java.sun.com/products/jini/>

specs/transaction.pdf

[22] Gang Ao. "Software hot-swapping Techniques." Technical Report SCE-98-11, Systems and Computer Engineering, Carleton University, December, 1998.

[23] E.Gamma, R. Helm, R.Johnson, and J. Vlissides. "Design Patterns, Elements of Reusable object-Oriented Software" Addison-Wesley Publishing Company, 1995.

[24] F. Ning. "S-Module Design for Software Hot Swapping Technology." Technical Report SCE-99-04, Systems and Computer Engineering, Carleton University. May, 1999.

[25] J. Bacon, "Concurrent systems," Addison-Wesley, 1997 2/e.

[26] P.A. Bernstein, and E. Newcomer, "Principles of Transaction Processing," San Francisco: Morgan Kaufman, 1997

[27] S. Oaks, and H. Wong, "Java Threads," O'Reilly, 1999 2/e

[28] A. Holub, "Programming Java threads in the real world, Part 8," available at URL:
http://www.javaworld.com/javaworld/jw-05-1999/jw-05-toolbox_p.html
