A Framework for Live Software Upgrade

by

Lizhou Yu

B.Sc., Beijing College of Economics, China, 1993

A Thesis Submitted in Partial Fulfillment of the
Requirements for the Degree of

MASTER OF SCIENCE

in the Department of Computer Science

We accept this thesis as conforming
to the required standard

[redacted]

Dr. G. C. Shoja, Co-Supervisor (Department of Computer Science)

[redacted]

Dr. H. A. Müller, Co-Supervisor (Department of Computer Science)

[redacted]

Dr. M. Storey, Departmental Member (Department of Computer Science)

[redacted]

Dr. W. W. Wadge, Departmental Member (Department of Computer Science)

[redacted]

Dr. K. F. Li, External Examiner (Department of Electrical and Computer Engineering)

Supervisors: Dr. G. C. Shoja and Dr. Hausi A. Müller

Abstract

Service availability is a critical requirement for safety and mission-critical software applications. Software applications usually undergo many changes during their lifetime. These changes are generally for enhancement, restructuring and correction of software in a production environment. The running application must be kept up to date with the newest version of the software. Costs, safety and feasibility become three major factors in the deployment of such non-stop applications. This thesis describes mechanisms designed and implemented to provide an easy, reliable, and cost-effective live software upgrade. To enable runtime addition, replacement and removal of modules, we present a dynamic software upgrade framework, consisting of four parts: a dynamic architecture and communication model, a reconfiguration management tool, an upgrade protocol, and an upgrade technique.

In the framework, the ability to make runtime modifications is considered at the software architecture-level. The dynamic architecture and communication model makes it possible for software applications to add, move, and hot swap modules on the fly. The transition scenario is specified by the upgrade protocol. The framework also provides the mechanism for maintaining state consistency. In order to ensure a reliable upgrade, a two-phase commit protocol is utilized to implement the atomic upgrade transactions. In addition, a command line interface in the framework facilitates the reconfiguration management of software applications.

The framework allows the target modules to be safely upgraded at runtime without disturbing other parts of the application. A simulation study of the proposed framework was carried out for live software upgrade of several practical applications. The results of the simulation are presented.

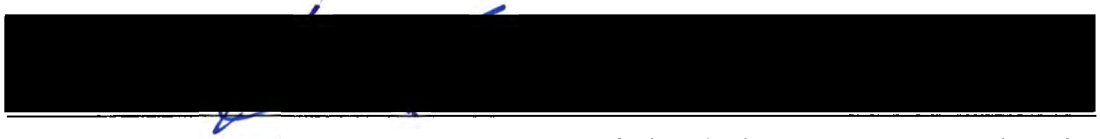
Examiners:

Dr. G. C. Shoja, Co-Supervisor (Department of Computer Science)

Dr. H. A. Müller, Co-Supervisor (Department of Computer Science)

Dr. M. Storey, Departmental Member (Department of Computer Science)

Dr. W. W. Wadge, Departmental Member (Department of Computer Science)



Dr. K. F. Li, External Examiner (Department of Electrical and Computer Engineering)

Table of Contents

List of Tables

List of Figures

Acknowledgments

I would like to take this opportunity to thank my co-supervisor, Dr. G. C. Shoja, who inspired me to work on this thesis topic and provided me with continuous guidance and endless patience. I can always get valuable suggestions and great help whenever I needed.

I am grateful to my co-supervisor, Dr. H. A. Müller for providing me an opportunity to enroll M.Sc. program, and helping me improve the thesis. His generosity, advice and support are gratefully acknowledged.

In addition, I would like to thank Dr. A. Srinivasan for providing me an opportunity to work at Nortel Networks, Ottawa. His support and encouragement are important for development of thesis. Moreover, I would like to thank Nortel for funding the System Availability project and providing their research facility.

Many thanks go to the members of Panda group, Rigi group, Nortel, and other friends who give me great help during critical time. It is impossible for me to list all their names.

Finally, I would like to thank my wife Sherry Zhang for always being there with me during the entire of M. Sc. program. Without her sacrifice and encouragement, this would not have been possible.

Chapter 1

Introduction

The demand for continuous service in safety- and mission-critical software applications such as Internet infrastructure equipment, Internet appliances, aerospace applications, military defense systems, telecommunication applications and medical products, is expanding. The dynamic software upgrade techniques, which are deployed for on-line maintenance and upgrades, can meet the demand for high levels of system reliability, availability and serviceability.

Software applications usually undergo changes during their lifetime. These changes are generally for enhancement, restructuring and correction of software in a production environment. The running application must be kept up to date with the newest version of the software.

For these safety- and mission-critical applications, it is unacceptable to shutdown and restart the system during a software upgrade. Monetary loss, interruption of service and damage can be caused by the traditional upgrade approach. For example, thousands of requests for telephone connection may be dropped during the down time of a telephone switching system.

This thesis explores various alternative mechanisms for run-time application changes. It illustrates a Dynamic Software Upgrade Framework, which gives a systematic and practical solution to on-line software change.

1.1 Justification

Despite advances in software design techniques, there are nearly no software applications that can be designed once and run without any change during their lifetimes. The evolutionary change of software is unavoidable due to changes in the environment or in the application requirements that cannot be entirely predicted during the design, or due to the bug-correction or enhancement of functionality.

1.2 Granularity of upgrade

The upgradeable components in the dynamic software upgrade include module, function, process, object, file, database, etc. These components in evolving applications should be abstracted as black box suitable for upgrade. The objective of on-line software upgrade is to be able to add, remove or replace any relevant components without significantly affecting other parts of the application. In other words, rather than the whole application being upgraded at once, only particular components are selected for incremental upgrade at a time and others are ignored.

1.3 Requirement in application change

The software application must run continuously during the update process. All the changes behind the scene are transparent to the user. The availability of the system can be defined as

System availability = MTTF / (MTTF +MTTR).

MTTF: meantime to failure of the system

MTTR: mean time to repair of the system

In the network communication industry, the criteria for high availability requires that the system operate 24 hours a day, 7 days a week with 99.999% uptime. Reducing MTTR is one way to increase high availability. MTTR includes the time taken for scheduled maintenance or repair of software. On-line software upgrades can significantly

reduce reinstallation time and the time taken for removing faulty components and uploading new components.

On the other hand, in order to ensure minimal interruption of service in an application, a fast, reliable, and atomic upgrade should be deployed in live software upgrade.

1.4 The procedure

The whole procedure of dynamic software upgrade can be divided as follows:

1. Prepare for the change
    - Make a plan of the upgrade and assess changes in application complexity that might result from a proposed upgrade.
    - The change scope should be defined. The administrator needs to identify components requiring change to achieve the desired functionality. Dependencies among components should be considered within the change scope as well.
2. Upgrade Management
    - Decide the sequence of changes. Choose suitable techniques for run-time upgrades and to keep the application integrity.
    - Anticipate the failure of an upgrade and prepare a rollback recovery during operation.
    - Use some configuration tools or command line interface to execute the upgrade.
3. Monitor the change at run-time

    The application should be returned to its pre-upgrade configuration if the new component is faulty.

1.5 The challenges

A lot of complexity and challenges are involved in dealing with upgrading non-stop applications. The differences between the new component and the old one may include its functionality, interface and performance. Only the selected components can be changed while the other parts of the application continues to function without change. A dynamic software structure is needed to facilitate the evolution through partitioning, modularization, dynamic configuration, and dynamic binding. It is important to safeguard the software application's integrity when changes occur at run-time. A run-time software upgrade cannot be done at any time, since it may halt or crash the application. The techniques for run-time upgrade are quite dependent on the operating system on which the application is running, and the programming language in which the application is written. The ability of dealing with failure to do upgrade transactions will influence the applicability of the live software upgrade.

1.6 The approach

To meet the requirements of reliability, evolution and maintainability for non-stop applications, this thesis describes a new and integrated solution that can upgrade software modules at run-time in an easy, reliable and efficient way. In order to solve the problems listed above, the proposed framework addresses four main areas: dynamic architecture and communication model, reconfiguration management, the upgrade protocol, and the upgrade technique. The framework can be used to on-line upgrade multi-task software applications, which provide multiple, mission-critical services.

We believe that the dynamic architecture and the communication model provide a foundation for run-time software evolution. The ability of run-time modification is investigated from an architecture level in the framework. Indirect addressing, the name service, the publisher and subscriber communication model, and the version-control repository form a unique dynamic software upgrade architecture.

The command line interface is employed in reconfiguration management to facilitate the execution of upgrade. It defines a set of transaction operations. The administrator can easily upgrade individual modules or a group of modules at run-time by sending the configuration commands.

The upgrade protocol defines the scenarios of modules addition, removal and replacement at run-time. The module-proxy is designed to synchronize the state between the replaced module and the replacing module. A practical strategy for the module to reach stable state is illustrated in the framework.

The two-phase commit protocol is introduced to ensure safe, reliable and atomic upgrades. Moreover, the concurrent upgrade mode is implemented to minimize the down time of service.

Object-oriented techniques, design patterns and dynamic link libraries are utilized to facilitate the implementation of the framework, including loading of the modules and the creation of the instances, etc.

1.7 The remainder of this thesis

Chapter 2 describes various related works from different points of view. It also discusses the advantages and disadvantages for those approaches deployed in run-time software upgrades. Chapter 3 provides a dynamic software upgrade framework. The overview of the architecture and basic building blocks, the methodology of run-time upgrade policy, the two-phase commit protocol, the concurrent upgrade mode and scope change are elaborated. Chapter 4 illustrates the implementation of components in the framework. Various strategy and methodology, the queuing analysis, design pattern and concurrent control, state transition, class diagram and object interface are addressed respectively in detail. Chapter 5 demonstrates an application developed by the framework. Finally, Chapter 6 summarizes the contributions of the thesis and suggests some future work.

Chapter 2

Related Work

Live software upgrade is a relatively new field and not many practical systems have been implemented. In this chapter, some of the work, which is closely related to the live software upgrade, is reviewed.

2.1 Hardware-based approach

In a primary-standby system, two devices run the equivalent program and back up each other. As a result, it is easy to achieve continuity of service with this architecture. To perform the update, the first device is stopped at a safe point in the program and simultaneously the second is started up. After the first one is upgraded, it takes the roles of second device, so the second device is taken off-line and is ready to be upgraded as well. The principal disadvantage of this technique is its substantial cost. Redundant device and duplicated running software are needed for this kind of systems.

For example, in telecommunication industry, Virtual Redundant Router Protocol (VRRP)[1] can be used not only for high availability but also for software upgrade. When an application is scheduled for upgrade or the master router is down, the backup router is forced to take over the responsibility from the master and redirect the traffic to itself. So service availability can be achieved. Because the application upgrade is processed on one router by one router basis, the whole system in the off-line router can be shut down without problems. The new program can be downloaded and installed entirely. And the

system can be rebooted again. The state machine model for each router is shown in Figure 2.1. In this case, VRRP needs two running systems and two physical devices.

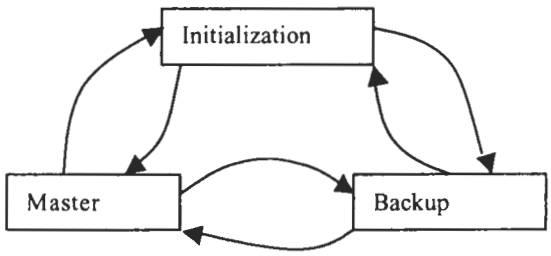


Figure 2.1 State machine model for each router

2.2 Component -based dynamic architecture

Darwin [2] proposed by Jeff Kramer and Jeff Magee, is a configuration language for describing dynamic architecture. It is a declarative language, which is intended to specify the structure of distributed systems composed from diverse components using diverse interaction mechanisms. It separates the description of structure from that of computation and interaction.

Darwin allows distributed programs to be specified as a hierarchic construction of components. Composite component types consist of the primitive computational components. And these components in turn can be configured into more complex composite types. Components interact by accessing services. Each intercomponent interaction is represented by a binding between a required service and a provided service.

By using Darwin configuration notation, the system administrator can configure the software components forming a distributed service. Composite components and systems are formed in Darwin by declaring instances of components and binding the services required by one component to the services provided in another.

2.3 Dynamic object–based approach

Dynamic object technology [5][6] uses the techniques of reflection, meta-data, and meta-object protocols. These techniques permit uniform coding, which can encapsulate the interfaces to heterogeneous data and legacy systems. It is possible to use dynamic object capabilities in the creation, maintenance, and enhancement of software products. In terms of the product maintenance and enhancement, dynamic objects allow software updates and bug fixes, and make it possible to migrate incremental changes into deployed applications gracefully.

The ***Common Lisp Object System*** (CLOS) [7][8] is one of several dynamic languages, which have three basic features:

- Both dynamic (run-time) and static (compile-time) typing
- The behavior of functions and other operations can be changed (or created) at run-time
- Reflective and object-oriented programming language

CLOS includes a meta-object protocol, which was originally defined as supplemental interfaces to programming languages that give users the ability to incrementally modify the language's behavior and implementation. The meta-object protocols allow users to adjust the design and implementation to suit their particular needs, and allow programmers to tailor the system to meet their particular needs.

Function in Common Lisp is first class. A "function factory" can be constructed which returns function objects as values. Thus, a program can produce the functions it needs at run time, making different functions for different situations.

Function *load* can be called by any Lisp program. If the file being loaded contains new definitions of existing functions, or classes, these new definition are added and made available to the application. Moreover, existing definitions are automatically replaced with new ones.

CLOS allows classes to be redefined in running programs, and the system automatically updates existing instances of the changed class to conform to the new definition. Bug fixes or new functionality can be installed without stopping the application by using Common Lisp.

JAVA [9] is also a dynamic object-oriented programming language. It supports dynamic loading of any class. Run-time Type Identification (RTTI) and Run-time Reflection which are supported in Java can determine the precise type of an object. These features enable the incremental change in a software program.

There are some restrictions in these dynamic languages. The main drawback of dynamic languages having meta object and run-time reflection is slow performance. For instance, every function invocation must be bound during run-time in Common Lisp. Moreover, the application needs to be written entirely in the dynamic language to benefit from dynamism.

2.4 Procedure-level dynamic updating

The ***Procedure–Oriented Dynamic Updating system*** (PODUS) [10][11], was developed at the University of Michigan and later enhanced at Bellcore. Each existing procedure can be replaced by a new procedure during execution time. Two basic requirements should be met. First, the program must be written in a top-down manner so that a logical call graph could be formed. Second, the data accessed by several different procedures are accessed through abstract data types. Procedures are updated only when they are inactive.

PODUS's architectural model consists of a very large, sparse virtual-address space. The large address space is partitioned into a number of version spaces. Within each version space, the code for a specific version of a program and its static data are stored, along with a binding table. The procedures in that version space use the binding table. The binding table provides a method of associating a procedure's name (the

version-independent specification of the task that the procedure is supposed to perform) with its address (the version dependent implementation of how the procedure performs its task).

During an update, PODUS identifies active procedures by examining the state of the program's run-time stack and procedure-call graph. PODUS must look at both syntactic and semantic dependencies in determining when an update is possible. When a procedure P_{old} is updated to P_{new}, the program is reconfigured so that when old versions of procedures call P_{old}, a user-written interprocedure maps this call into a call to P_{new}. The interprocedure will be called with P_{old}'s specifications and will call P_{new} with P_{new}'s new specifications. This will continue until all the old procedures that call P_{old} have been updated. By encoding a machine number into the space address, it can scale the updating system into a distributed environment.

PODUS focuses on updating the implementation of procedures, however, the granularity of replacement is obviously too small. As a result, a procedure-based approach can't deal with changing modules within the system. The approach also assumes a fixed top-down programming style, which is not suitable for changing the structure of the system.

2.5 Process-based approach

Deepak describes an approach [12][13] to modeling change at the process level for a simple imperative programming language. He considered the problem of on-line version change, in which the software is updated while being executed. He shows that an on-line change from an old to a new version needs to maintain application integrity, and the new version of the function must be a superset of the old version so that the change is transparent to the caller.

At first, the system creates a new process with a new version of the software and monitors the execution of the old version. After certain conditions are met, it transfers the

state of the old process and the control to the new process. After the state transfer is over, the old process is killed and the new one is continued. For a valid on-line replacement, the system has to ensure that the state is transferred at a time when none of the routines is on the stack. To check if state can be transferred at some point, the shell determines the set of routines, which are on the run-time stack of the process running the old version. When the last routine on the stack that is to be changed returns, the process incurs an illegal instruction trap. Furthermore, the shell, which has been waiting for this event to occur, is notified. At this point, the stack is guaranteed to contain no routines, which is to be changed. State transfer takes place at this time. To transfer the state, the shell first makes the process running the second version of the program expands the data and stack space to occupy the same space as the first process. The replacement module then copies the data and stack of the first process onto the second one. The machine registers are copied next.

Simplex [14] is both a process-based and an analytic redundancy based approach. In the process-based approach, the replacement unit in Simplex is defined as a process, which encapsulates a set of computation activities and makes address protection.

The process-based approach is restricted when a replaced module is considered as a heavy-weight process. However, it cannot be used when all the modules are running as a heavy-weight process, and each module is associated with one thread context. Moreover, the approach requires copying data in stack and machine registers from the old process to the new process, and inserting some exchange points in advance. Therefore, it involves much overhead and complexity.

2.6 Analytic redundancy based approach

Jonathan proposes a framework ***Hercules*** [15] for upgrading system components. Instead of removing the old version of the component, the system keeps multiple versions of a component running. Doing so maintains system integrity and correctness even in the

presence of newly introduced errors. It is easier to roll back the upgrade if it turns out that the new version of the component does indeed break some existing functionality. It provides the highly reliable upgrading of components, by keeping existing versions of the component running and only fully removing the old component when a determination is made that the new component has fully satisfied its role.

In the analytic redundancy based approach, ***Simplex*** [16][17] deals with safely upgrading real-time control software without shutting down the normal operation. Replaceable units communicate with each other via a real-time group communication facility. And real-time process management primitives are built on a generalized rate monotonic theory. There are two ways of managing replacement operations: Supervisor-triggered and self-triggered. The basic software unit that packages a distinct application function is the subsystem unit. A subsystem module is used to implement a distinct application function. A subsystem module consists of specialized replacement units in the form of an application *independent module management unit* and several application units (as shown in Figure 2.3).

The *module management unit* is a replacement unit with functions that are designed to carry out process management and the upgrade operation. A typical set of application units consists of the baseline unit, the new unit and the safety unit. The *baseline unit* runs the existing baseline software; *a new unit* represents the intended upgrade for existing software, and the *safety unit* implements a safety controller, system performance and safety monitoring functions. After replacement, the *baseline unit* and the *new units* are both introduced into the system. Both outputs are compared, and the output from the new unit is used. If the system under the *new unit*'s control violates the operation module, the system will mark the *new unit* as faulty and uses the safety controller's output until the system returns to the operation state.

Hercules and *Simplex* permit safe on-line upgrading of software in spite of residual errors (breaking the function rules) in the new components. Analytic redundancy

facilitates an extensive testing for reliable incremental evolution of safety critical systems. It focuses on how to rollback when a new unit doesn't satisfy explicit performance and accuracy requirements after replacement. However, it does not illustrate well how to deal with the failure during the upgrade transition.

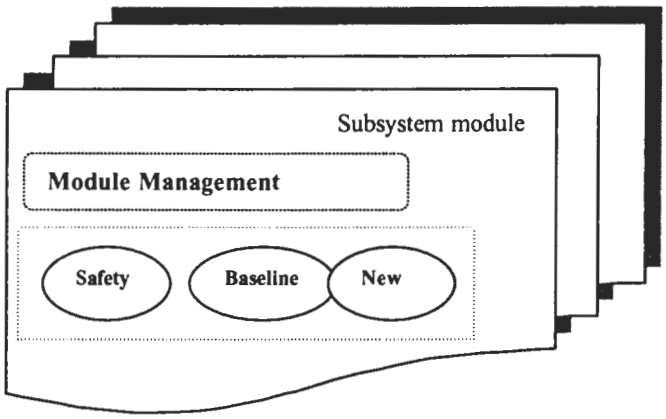


Figure 2.3 Replacement Unit

2.7 Summary

We have shown how different researchers have tried to solve the problems of live software upgrade from different levels and different points of view. Updating on primary-standby system relies on redundant hardware and software. Dynamic architecture and dynamic language facilitate separation of component communication from computing, and they enable reconfiguration and the incremental evolution of application software. The process-based and procedure-based approaches achieve run-time change through indirection of function call and state transfer between processes. Analytic redundancy enables on-line testing and reliable upgrading. None of these approaches has addressed all of the problems adequately. In the next chapter, a dynamic software upgrade framework is proposed which provides a new and integrated solution.

Chapter 3

A Dynamic Software Upgrade Framework

This chapter proposes a dynamic software upgrade framework, presents its building blocks, and addresses issues on state persistence and consistency, scope change, run-time upgrade policy, atomic transaction and concurrent upgrade mode.

3.1 Overview

A non-stop software application must be able to add and remove software components at run-time in order to add new services, maintain the functionality, and reconfigure the system. A **dynamic software upgrade framework** provides the flexibility for software applications to be evolved, adapted and tuned in a rapidly changing operating environment. In addition, it maintains consistency of state during the change of this application, minimizes the side effects due to downtime, ensures safe upgrade transactions and ensures that the deadlines are met for time sensitive applications.

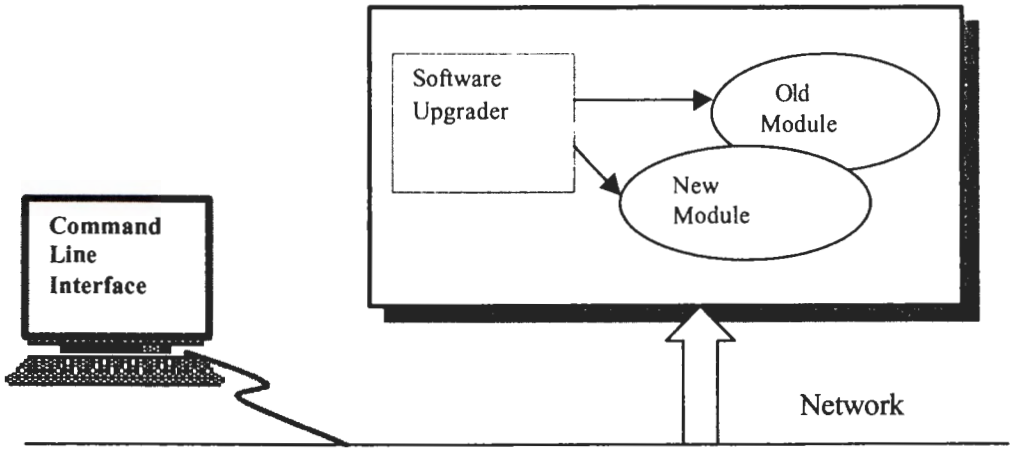


Figure 3.1 Upgrade of the application

Usually, the running application consists of many modules. Figure 3.1 shows that an administrator can locally or remotely reconfigure the software application through a command line interface. The upgrader-enabled application responds according to external commands to load, unload, disable, enable and replace modules.

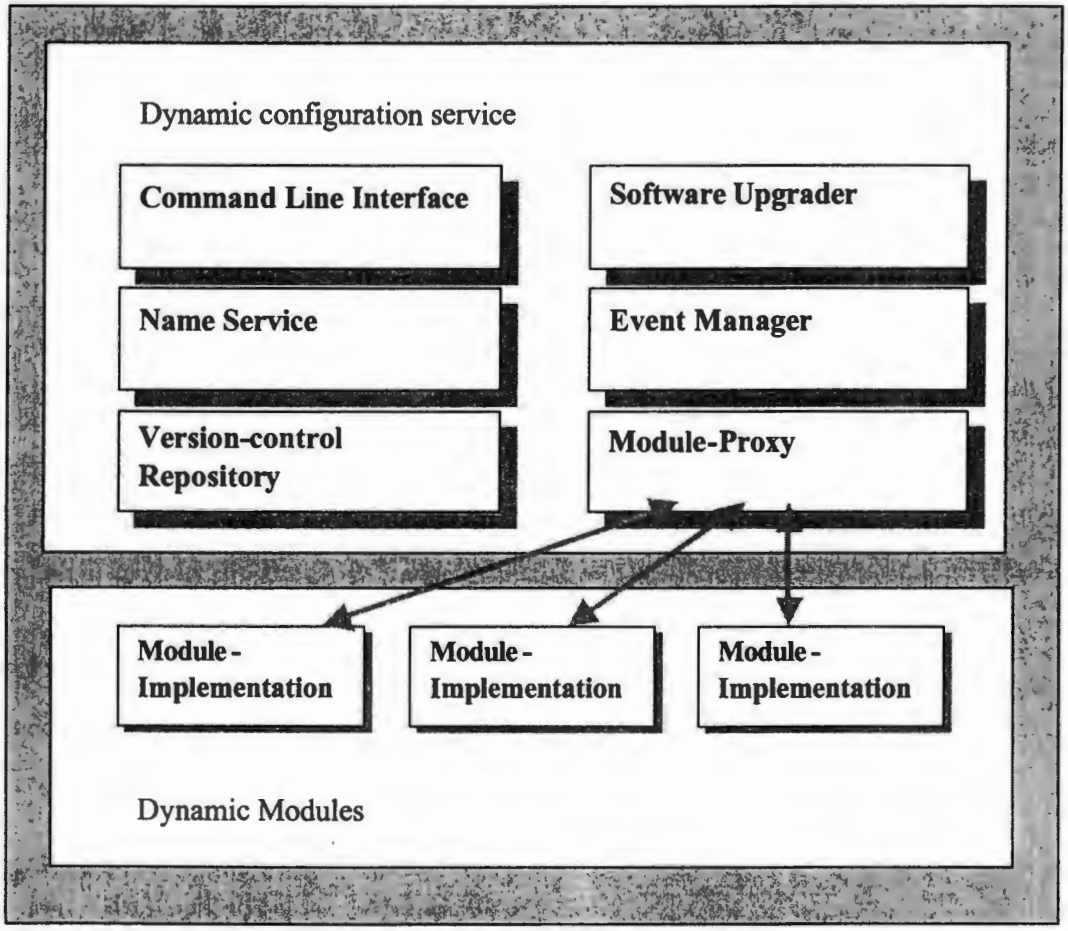


Figure 3.2 A dynamic software upgrade framework

Figure 3.2 depicts a **dynamic software upgrade framework**, which provides a general and integrated solution to deal with upgrading non-stop applications at run-time. It can be divided into two parts. One part is the dynamic configuration services. These include a Command Line Interface, a Software Upgrader, a Name Service, an Event Manager, a Version-control Repository and a Module-Proxy.

The other part is the dynamic modules, which consist of module-implementations. Dynamic modules are those upgradeable components, which can be disabled, enabled, loaded, unloaded, and hot swapped.

3.2 The goals of a dynamic software upgrade framework

A **dynamic software upgrade framework** is designed to solve problems of on-line software upgrade and achieve the following goals:

1. *Enable multi-tasking software applications to achieve function enhancement and fault avoidance.* Evolution of software is inevitable due to changes in the environment or in the requirements that cannot be entirely predicted during the design, or due to misbehaving of software modules. The traditional method of software upgrade, consisting of procedures for stopping the whole application, installing the new version, and finally resuming application execution, is no longer suitable for mission-critical zero-down-time applications. Our solution is to build an application where parts of the application can be replaced on the fly, while the rest of the application continues functioning. In this way, the change for the software application should be easy, low cost and safe.
2. *Ensure atomic upgrade transaction without influencing other running modules.* The upgrade transition should only include affected module components, while other components should still work properly. Moreover, all the upgrading actions should be considered as atomic operations. An atomic action either performs its computation fully or does not perform the computation at all. This is what is called "all-or-nothing". In the case of a failure, the transaction will be aborted. For example, when the replacement of an old module fails, the old module should be fully recovered.
3. *Minimize upgrade time and enable concurrent upgrade.*

It is important to enhance service availability by reducing maintenance time. It is very costly if an application is out of service for a long time. An appropriate strategy should be chosen in order to upgrade a group of modules simultaneously with negligible down time.

4. *Keep the application state consistent with the specific application requirements.*

   In some cases, changing software components without considering maintaining application state will result in the application inconsistency. Even worse, it could result in a program crash or some other unpredictable errors. However, keeping the state consistent during the upgrading transition can be a big challenge for non-stop applications.

5. *All the upgrade transaction can be initiated from the CLI (command line interface).*

   As discussed previously, the administrator decides when and how to upgrade the application. Upgrader-enabled application allows the administrator to reconfigure the application locally or remotely by CLI. The command interface should be designed for administrator to handle all the different situations.

6. *There is a run-time evolvable software architecture*

   The software applications should be initially designed for evolution. Dynamic communication and interaction between software components should be introduced during design. Module behavior should be dynamically changeable without much effort. Additionally, module interaction should be decoupled and well managed.

7. *Permit the incremental execution of reconfiguration steps in real time.*

   Application transition should be made gradually with time constraints. Within a certain time, affected parts in the application should be ready for upgrade. The timing constraints are important for a real-time applications in order to meet the demand of service availability.

3.3 Building blocks and basic scenario

3.3.1 Addressing the problem and division of modules

Usually, modules communicate with each other through message passing. During the communication process, synchronization has to be taken into account. In general, there are three combinations of synchronization: blocking send and blocking receive, non-blocking send and blocking receive, non-blocking send and non-blocking receive. The common combination is a non-blocking send and a blocking receive. In other words, the sender may continue after sending the message, the receiver is blocked until the requested message arrives.

In direct addressing, the sender needs to know a specific reference to the destination. However, after an existing module is replaced by the new one, re-linking other modules with the new one becomes a big issue. The alternative is indirect addressing. In this case, the messages are not sent directly from the sender to the receiver but to a well-known port called mailbox, which has a shared data structure consisting of queues that can temporarily hold messages. Using indirect addressing can decouple the sender and the receiver and gives great flexibility in dynamically updating the existing modules (as shown in Figure 3.3).

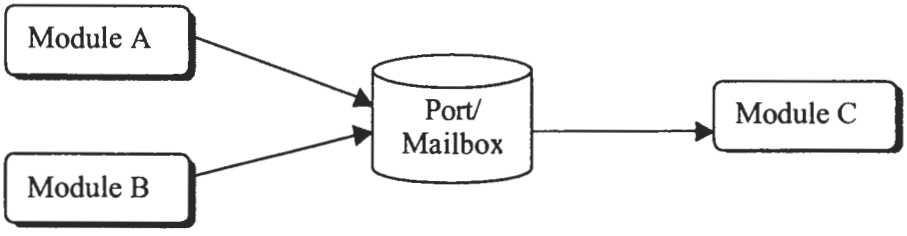


Figure 3.3 Intermodule communication

In order to extend indirect addressing, we split an ordinary module into two parts. One component is a module-proxy; the other is a module-implementation.

A module-proxy is used for minimizing coupling between modules. It prevents the implementation modules from directly referencing one another during communication. When a module-implementation is in the *Service* state, its module-proxy forwards all the incoming requests to it (as shown in Figure 3.4).

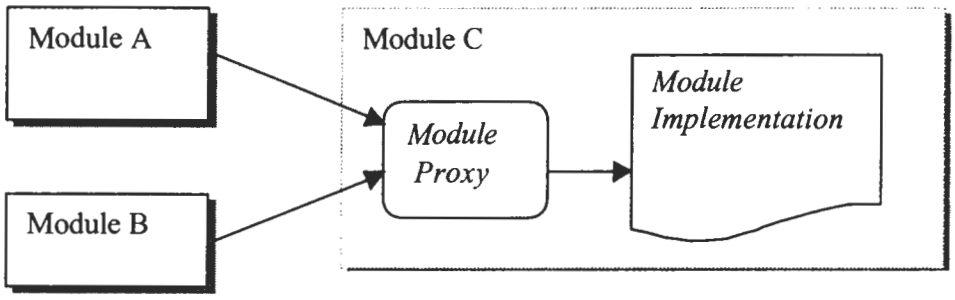


Figure 3.4 Communication between modules

3.3.2 Module-proxy

Module-proxy not only works as a well-known port but also coordinates on-line changes with its module-implementation.

1. It is given with a module name in the beginning. At run-time, it is dynamically associated with a module-implementation by the application. A module-proxy delegates its module-implementation and works as logical connector between two communication end-points. Moreover, it forwards all the incoming requests to its module-implementation.
2. In addition, it shall coordinate with a software upgrader to dynamically update its module-implementation. Thus, it can alleviate the burden of the software upgrader. Since the module-proxy always maintains reference to its current module-implementation. At the time of updating, the module-proxy will swap its reference like port migration, redirect the incoming messages to the new module-implementation, and meanwhile unload the old one. In case of failure in

upgrading or timeout, the module-proxy should enforce recovery, bring back the original module-implementation into its *Service* state, and clean up the new one.

3. The module-proxy is used for directing and monitoring request messages between module-implementations. This feature gives it the opportunity to control the consistence of the internal state. The module-proxy is in charge of transferring internal state between the old and the new module-implementation. During the upgrade transition, the incoming messages should be buffered and forwarded by the module-proxy to its new module-implementation.

3.3.3 Module-implementation

The module-implementation, as a dynamic module, encapsulates all the application specific implementation and runs as a task. Besides, it can be updated on the fly. It has the following characteristics:

1. It implements some application functionality to meet certain specification requirements. It also has to contain all the internal state information related to that application functionality. Moreover, it must provide a certain amount of functional behavior to participate in the run-time change.
2. It should be suitable for replacement and enhancement from the application point of view. To support run-time evolution, module-implementation must be packed in a form that can be loaded and unloaded dynamically under the run-time environment.
3. There is an input message queue associated within each module-implementation for asynchronous communication. Synchronous communication modes can be also supported by adding a reply queue.

4. The module-implementation should run as a task, which is associated with a thread context. It would internally dispatch functions based on the received message's type.
5. In general, the module-implementation can be divided into two kinds of modules: *passive modules* and *active modules*. *Passive modules* only wait for receiving requests from the outside and then serve them. On the other hand, *active modules,* not only receive requests but also initiate requests to the other modules.
6. The module-implementation should be bound with a module-proxy as its port is used for indirect addressing. When module-implementation is in a Service state, all the incoming requests are appended into its input message queue.

3.3.4 Version-control Repository

In a dynamic software upgrade framework, each module-implementation can be compiled as a module library, which is an upgradeable unit that can be unloaded and loaded at run time. It is very important for the software upgrader to have a library repository, which records and holds all the loaded module libraries within the application memory. Once the software upgrader loads a module library into memory, it registers the handle of the module library into a version-control repository so that the manger can use it to create an instance of that module or clean up the loaded module library according to different situations. Moreover, the repository can be used to validate new handles and avoid duplicate loading of the same module library.

3.3.5 Command Line Interface

In order to fully control operations during the application transition at run-time, the application administrator should be given permission to reconfigure a software application through a command line interface. In general, there is a remote mode or a local mode during application reconfiguration.

To maximize flexibility for application transition, the command line interface can fall into two categories according to its functionality. The first category deals with application transition for only one module. The second category deals with simultaneously upgrading a group of modules that depend on each other.

3.3.6 The dynamic communication architecture

Only exploring the upgrading mechanism is not enough for building a run-time evolvable software system. Having a dynamic communication architecture could address issues such as *scalability* - the application can grow larger in order to provide added power and functionality, *dynamic changeability* - changed modules can notify and multicast their changes to the rest of system, *dependability* - the modules' interaction and dependency can always be changed at run-time without much effort.

Publisher and subscriber communication model matches the above characteristics. It is an event-based architecture. A publisher can publish any event; a subscriber can subscribe to any interesting event and receive all event messages. At any given time, the publishers do not need to know the subscribers and vice-versa. Publishers and subscribers may also obtain and remove publication and subscription rights dynamically. A module can be both publisher and subscriber at the same time. Figure 3.5 shows that module *Timer* publishes "multicast" event and passes the event tag to the event manager. Event Manager is in charge of controlling the event table. In addition, Module A and Module B search for the "multicast" event tag and subscribe to it. Subsequently, the event manager puts modules *A* and *B* under the list of subscribers. Afterwards, when a "multicast" event happens, the Timer module will multicast an event message to all the subscribers. At that time, the *Timer* module doesn't know how many names have subscribed for the "multicast" event.

Modules can dynamically change their subscription of interests. When one module joins or is deleted from an event tag, it will not affect other modules. By holding

all the event tag information and centralizing the management of module communication, the publisher and subscriber communication model reduces the overhead of software evolution at run-time.

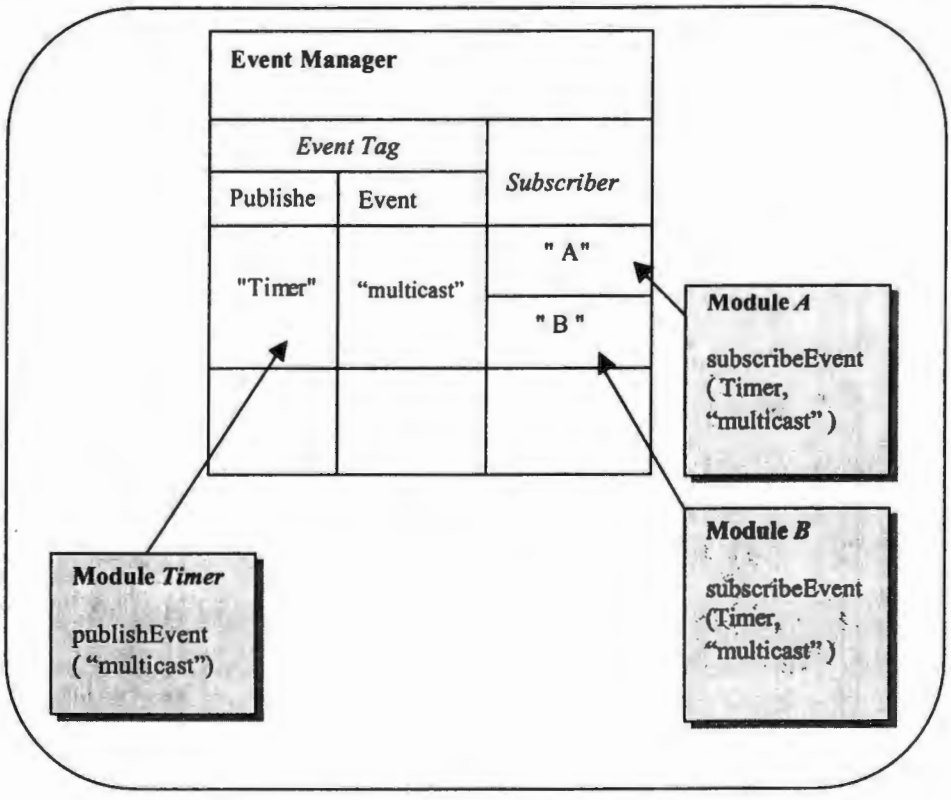


Figure 3.5 Publisher and subscriber event communication model

3.3.7 Event Manager

The event manager contains all the information on registered event tags and interested subscribers. Each event tag shows the name of the publisher and its published event type. Each Module has to get the handle of the event manager before publishing or subscribing to events.

The publisher needs to create an event tag with its module name and event type before passing it to the event manager. The event manager then appends the event tag into a new entry.

The subscriber needs to find the event tag first by searching for the name of the publisher and the event type. After that, it tells the event manager to subscribe to the event tag of interest. Then, the event manager puts the name of the subscriber under that event tag. The list of event tags and subscribers expands or shrinks dynamically.

After a publisher tells the event manager that it wants to notify its events now, the event manager should first look for all the subscribers under that event tag, and then get their module-proxies by searching the name service. Finally, the event manager sends the event notification in the form of an event message to all the subscribers.

3.3.8 Name Service

A dynamic software upgrade framework provides a uniform name service:

- Any module can be bound to any name.
- The name of a module is conventionally represented as a string.
- The name service can be used to bind the name of a module and resolve the reference to a module at run-time.

For example, Module "B" can obtain a reference to Module "A" by resolving the name of Module "A". Then Module "B" could use that reference as a port to send a message to Module "A". In order to reduce the direct addressing of modules, the notion of a module-proxy is introduced to delegate module-implementation. Instead of a reference to a module-implementation, the reference to a module-proxy is registered with the name service and bound to the name of the module. In this way, a module-proxy forwards all the incoming requests to its module-implementation. Removal and replacement of a module-implementation within that module becomes transparent to other modules.

Figure 3.6 shows that Module "A" and Module "B" both register with a name service. It includes the name of the module and a reference to its module-proxy. Module "B" and Module "A" can discover each other at run-time. Module "B" can only get a

reference to a module-proxy in Module "A" by resolving the name of Module "A". Changing the implementation of Module "A" can be done without influencing the activity of Module "B".

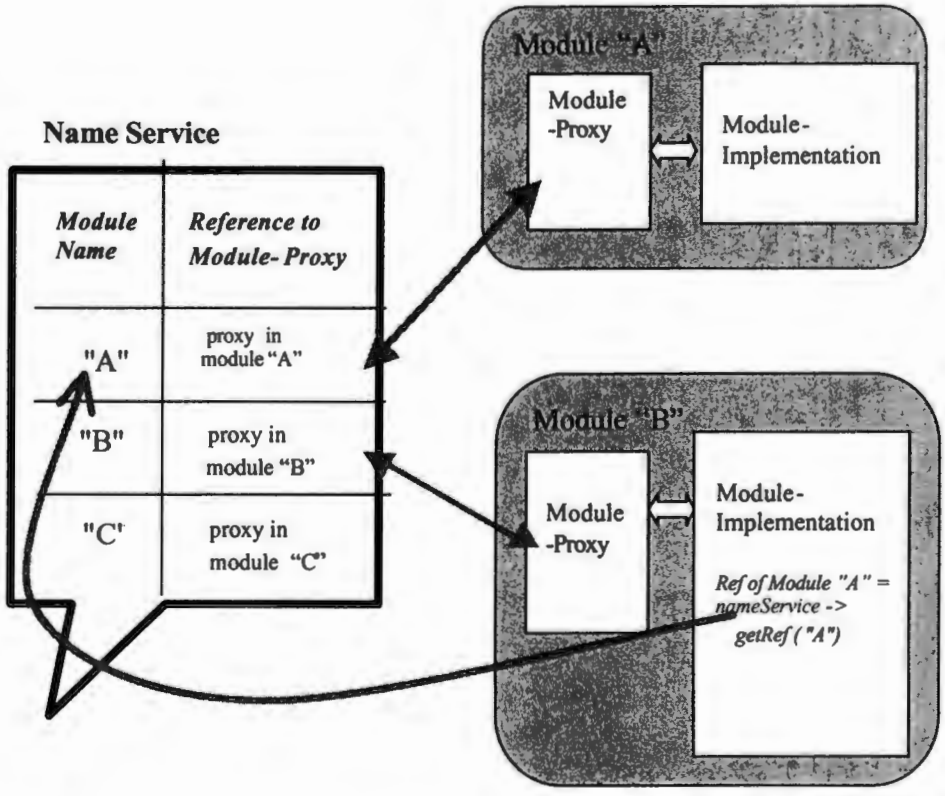


Figure 3.6 Dynamic Binding and Discovery

3.3.9 Software upgrader

Software upgrader plays an important role in a live software upgrade including the following tasks.

- The software upgrader is normally blocked, waiting for upgrading requests from the command line. Once the command is issued, the software upgrader is awakened and begins to execute different tasks such as disable, enable, load, unload and replace the module.

- If adding or replacing a module is required, the software upgrader is in charge of uploading the new module library into memory. Then it registers the handle of the library to the version-control repository.
- After uploading, it should create an instance of a dynamic module. Meanwhile, it registers a reference to a module-proxy within the name service, marks it as active and maintains the application integrity.
- If removal of a module is required, it should remove the module library from the version-control library and update the name service as well.
- During updating of a module-implementation, the software upgrader should coordinate with the module-proxy. The software upgrader makes the final decision for committing or aborting an upgrade transaction. If a failure occurs or a timeout is fired, the upgrader shall abort the transaction and ensure rollback.

3.4 Persistence and Consistency of state

3.4.1 The module state

The state of a module is always changed while the application is involved in different transactions. If one module is replaced, the new module must discover the state of the old module and perform the necessary actions to synchronize its internal state with that of the old module. It is important to determine which state should be transferred from the old one and what time is suitable for checkpointing. The application state can be very complex at run-time. It can be roughly divided into a stable and a transient states. In order to avoid overhead such as deep copy data of the stack and machine registers, the transient state should not be considered for checkpointing. Once a module goes into a kind of stable state, the module state can be checkpointed. Persistence of state enables a module to save its internal information on a stable storage, so that it can be recovered later.

3.4.2 Quiescent state

Preservation of application consistency requires that any replacement of software modules should yield a correct configuration and maintain the consistency of affected modules, which make up the application. Configuration management must give the affected part of an application the opportunity to reach a consistent state before a change is performed. The configuration of the application does not force changes but waits for a software module to reach a consistent state, which is called the *quiescent* state [18]. The module to be replaced reaches a quiescent state if

- it is not currently initiating any request and
- it is not currently engaged in serving a request.

Since the quiescent state is reached, it is the right time to checkpoint the state of the target module. To keep state frozen, the target module should be entirely isolated from the other modules that can start new requests capable of causing a state change on the target module. Under these circumstances, it is quite a high burden for the application to notify other modules. Two strategies can be applied to solve this problem.

In the first strategy, all new incoming request messages are buffered into a message queue while the target module is upgraded. All the buffered messages are passed to the replacing modules later. As a result, no request message gets lost during updating.

The second strategy is to use the publisher-subscriber communication model. For example, the replaced module notifies all the subscribers when a replacement event occurs. Other modules stop sending requests to the replaced module as soon as they are notified. The idea is that each module takes care of itself and when an event happens, it simply multicasts the event.

The module-proxy of each module should wait and coordinate its module-implementation to reach its quiescent state. Its responsibility is to:

1. Buffer the future incoming requests in the format of a message while the software upgrader requires a replacement.

2. Make its module-implementation inactive to stop initiating any new requests to others.
3. Send a termination message to its module-implementation and wait for the module-implementation to reach stable state.

Therefore, the module-implementation finally reaches a quiescent state since it does not issue any request to others, and it is not engaged in serving a request any more.

On the other hand, the module-implementation needs to be aware of when an upgrade event occurs. Additionally, it should tell its proxy when it arrives at a stable state. The content and scope of a transferred state should be specified by the software application.

3.5 Scope Change

Scope change is the extent to which the software modules in an application are affected by an on-line change. Since a message protocol is defined in communication peer, changing of the message protocol on one side must affect the other side. Because of compatibility, both modules become replaced modules when their message protocol is changed. Such a dependency on a message protocol decides the scope of target modules. Thus, the administrator can easily change the scope by declaring a dependency between modules.

However, a problem arises when a new version of a message protocol is introduced, and two versions of a message protocol may both be used in communication between software modules in the application.

We assume that the administrator knows the dependency graph between modules. A strategy can be used to adjust the scope change. In order to achieve backward compatibility, two versions of the message protocol should both be accommodated by the replacing module. This will continue until all the modules that use the older version of the message protocol have been updated. In this way, scope change becomes minimal.

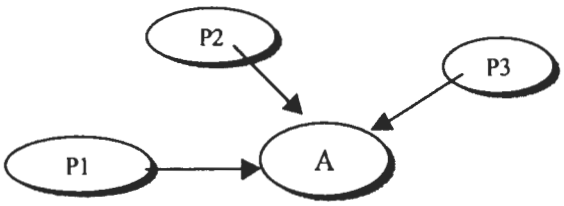


Figure 3.7 Scope change

Figure 3.7 illustrates how scope change could be determined in a live software upgrade. In a software application, Modules *P1*, *P2*, and *P3* communicate with Module *A* using the same message protocol. Now, Module *P3* and *A* decide to use a new message protocol instead of using the old one for some reason, such as a functional enhancement. According to our strategy, the new version of Module *A* is implemented to have the ability to deal with two versions of a message protocol. In this case, the old version of a message protocol used by Module *P1, P2,* and a new version used by Module P3 will be supported by the new version of Module *A*. In this way, the application can keep its integrity after target Module *A* and *P3* are upgraded to their new versions.

3.6 Run-time upgrade policy

The *run-time upgrade policy* is an important part of a live software upgrade. In order to deal with the replacement, creation and removal of a module, the following protocols are introduced: ***module replacement protocol, module creation protocol, and module removal protocol***.

3.6.1 Module Replacement protocol

Module Replacement protocol describes how to make an on-line replacement of a module. It can be divided into three phases: the uploading phase, the replacement phase, and the cleaning phase.

- **Uploading phase** (see Figure 3.8)

1. The software upgrader receives a *replacement* command from the application administrator.
2. The software upgrader should look up the command message, search the name service, and validate it if the updated module has been activated and uploaded. Otherwise, it aborts the command.
3. After the validation check, the software upgrader should upload a new module library into the memory according to the command message, and register the handle of the library with the version-control repository.
4. Then the software upgrader creates an instance of a dynamic module; besides, it notifies the module-proxy that the current module-implementation will be upgraded.

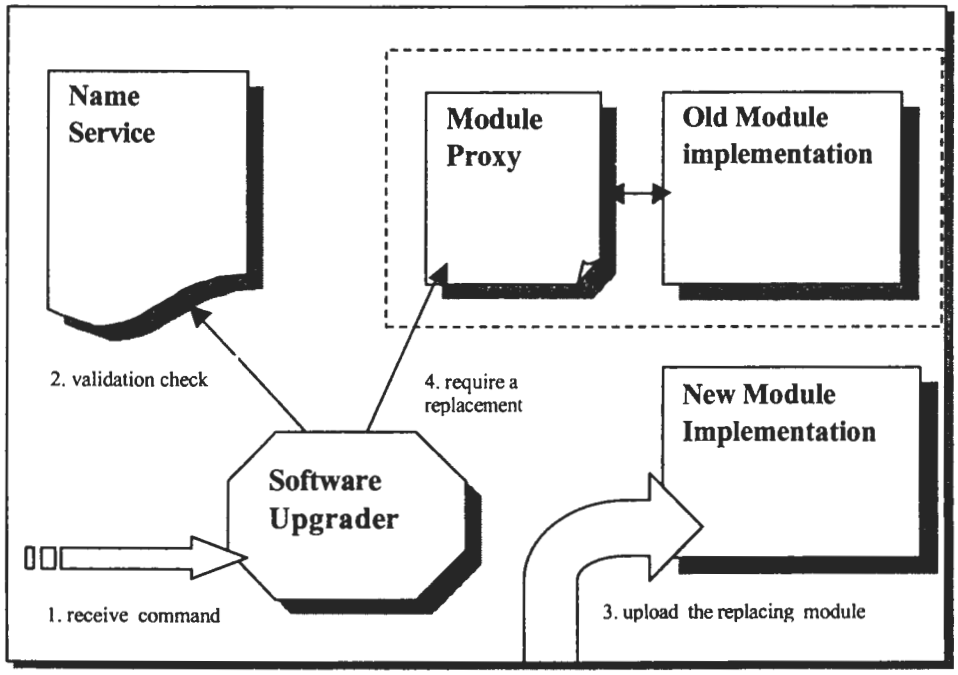


Figure 3.8 Loading phase

- **Replacement phase** (see Figure 3.9)

1. The module-proxy helps its module-implementation to achieve a quiescent state, and waits for a notification from its implementation.
2. When reaching its quiescent state, the module-implementations call the proxy back.
3. The module-proxy asks its module-implementation to checkpoint the state, and store it into a particular storage.
4. Then the proxy passes the handle of the storage to the new module-implementation and lets it recover the state stored. So the new module-implementation gets the chance to synchronize its state with that of the old one.
5. The proxy is associated with the new module-implementation dynamically. Finally, the proxy activates the new module-implementation and redirects the incoming requests to it.

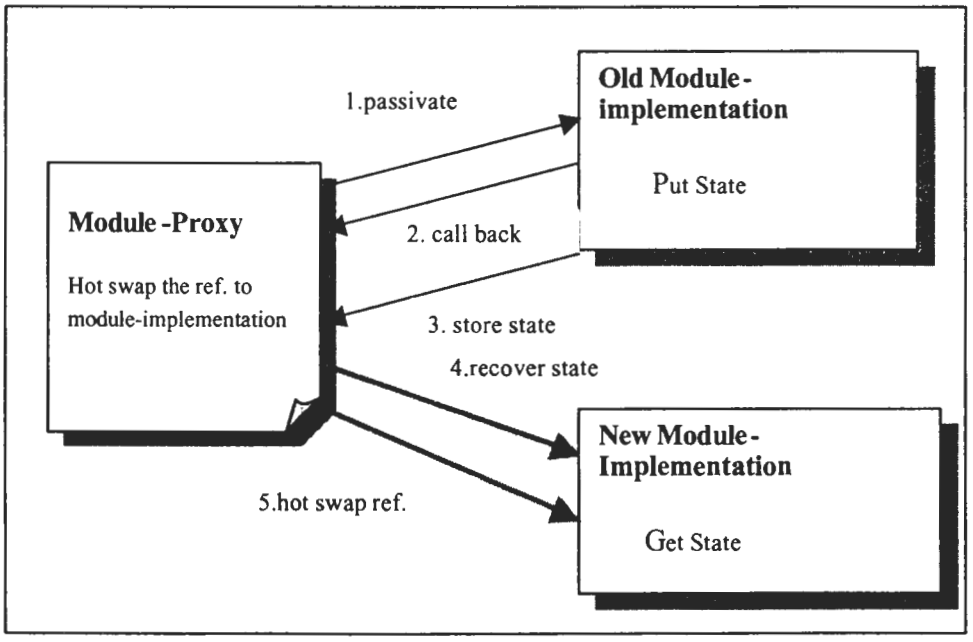


Figure 3.9 Replacement phase

- **Cleaning phase** (see Figure 3.10)

1. The proxy unloads the old module-implementation from the memory.
2. The software upgrader unregisters the handle of the old module library from the version-control repository.

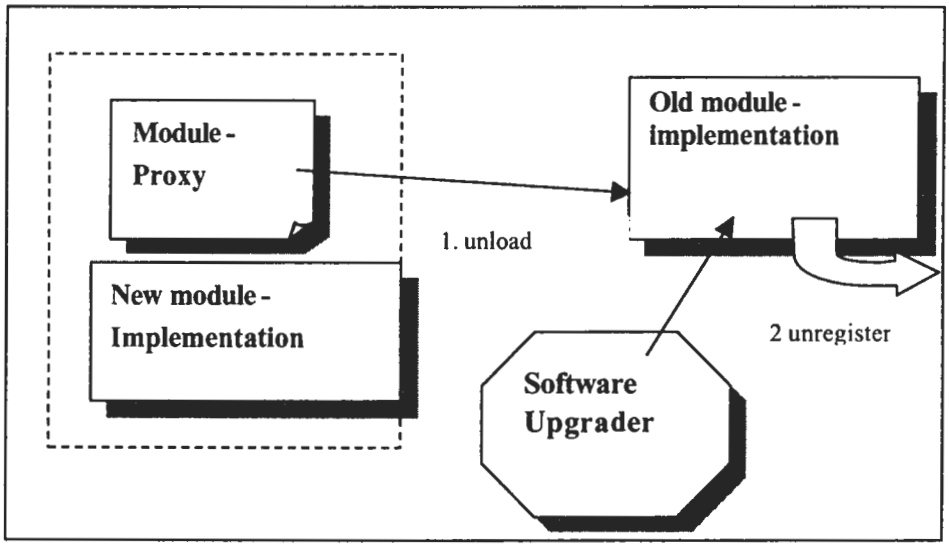


Figure 3.10 Cleaning phase

3.6.2 Module addition protocol

The ***module addition protocol*** is simpler than the *module replacement protocol*. The process of module creation can be split into two phases: the uploading phase and the creation phase.

- **Uploading phase**

1. The software upgrader receives a *loading* command from an administrator.
2. According to the command message, the software upgrader looks up the name service and validates if the loading module has **not** been uploaded yet.

3. After a validation check, the software upgrader uploads a new module lbrary. Furthermore, the library handles are registered into the version-control repository.

- **Creation phase**

1. During creation, the software upgrader not only creates a new instance of a dynamic module but also creates a new instance of a module-proxy. The instance of a proxy, a delegate, is associated with the instance of a module-implementation. The software upgrader also binds the module-proxy with the created module's name and registers it into the name service.
2. The module-proxy and its module-implementation are enabled. The created module comes into service and is available to other modules.

3.6.3 Module removal protocol

The ***module removal protocol*** aids in the graceful removal of a module. The protocol is divided into two phases: the preparation phase and the removing phase.

- **Preparation phase**

1. The software upgrader receives a *removal* command from an administrator.
2. The name service is looked up for validating if the module involved is active and has been uploaded already.
3. Afterwards, the target module-implementation is disabled and made inactive by its module-implementation. Finally, the removing module goes into a quiescent state.

- **Removal phase**

1. The target module-implementation is deleted from memory. The module-proxy is disabled and unregistered from the name service. The module becomes unavailable to others in the system.
2. Finally, the software upgrader removes the library handle from the version-control repository.

3.7 Upgrade techniques

3.7.1 Atomic transaction protocol

The atomicity of the application transition requires that when a live software upgrade to a group of modules is completed, either all of its operations are carried out or none of them. In case of failure of operation, the transaction should be aborted. The failure of an upgrade transition may occur if any target module:

- fails to reach a quiescent state within a bounded time
- fails to extract or store its internal state.

In order to achieve atomic operation, the software upgrader, as a coordinator, shall communicate with all of the modules involved in the transaction. The ***two-phase commit*** protocol is designed to allow any module to abort its part of a transaction. Due to atomicity, if one part of a transaction is aborted, then the whole transaction must also be aborted.

The two-phase commit protocol consists of a voting phase and a completion phase:

- **Phase 1 (voting phase):**

  1. The software upgrader sends a `CanCommit` request to each module-proxy in the transaction. As a consequence, module-proxy requests its module-implementation to reach a quiescent state.

  2. If the replaced module can successfully reach its quiescent state within the required time constraints, the module-proxy replies with its (`yes`) vote to the coordinator. Otherwise, the proxy replies with a (`no`) vote.

- **Phase 2 (completion phase):**

  3. The software upgrader collects the votes.

     1) If there are no failures, and all the votes are `yes`, the software upgrader decides to commit the transaction by sending a `DoCommit` request to each of the involved modules.

2) Otherwise, the coordinator has to abort the transaction and announce `AbortTransaction` request to all the modules.

4. The target modules that have voted will wait for a `DoCommit` or `AbortTransaction` request from the software upgrader. Upon receiving the request, the module-proxy will take action accordingly.

For each module proxy, the accomplishment of `DoCommit` is to

1. Transfer the state from its old module-implementation to the new one.
2. Hot swap its reference to the new module-implementation.
3. Provide the new module-implementation with a queue that buffered all the incoming request messages during the change.
4. Delete the old module-implementation and release the resource that has been held.
5. Activate the new implementation and begin to forward incoming requests to its current implementation.

Comparably, the completion of `AbortTransaction` is to

1. Provide the original module-implementation with a queue containing all the buffered messages.
2. Remove the new module-implementation from memory.
3. Resume the old module-implementation and let it continue to provide service.

3.7.2 Concurrent versus sequential upgrade mode

When a group of modules is upgraded simultaneously and a two-phase commit protocol is applied, an appropriate strategy of an execution should be chosen to reduce the downtime of the services in an application. In the scenario shown in Figure 3.11, there are three target modules in a sequential upgrade mode. In the voting phase, the software upgrader sends a `CanCommit` request to one of the target modules, and waits

for the vote from this module. After voting, the software upgrader then sends a `CanCommit` request to a second module, and so on. After the upgrader gets votes from all the modules, it moves to the second phase. In completion phase, according to the final voting result, the application must commit or abort the transaction on a module by module basis. In this case, the software upgrader must spend time on waiting for one module to finish its task before sending the request to the next module. In this kind of sequential upgrade mode, the more modules are involved in a transaction, the longer the service downtime.

Instead of a sequential upgrade mode, a concurrent upgrade mode as shown in Figure 3.12 can be used to avoid the problem. The **master and slaves mechanism** is introduced to allow concurrent execution. At the beginning, the software upgrader, as a master, will spawn a slave thread for each target module. In addition, each slave, as a subordinate, is associated with a task and is in charge of upgrading a module. Then, in the voting phase, software upgrader can send `CanCommit` requests to its slaves and let them take the necessary action. Consequently, the software upgrader is blocked waiting for a final voting result. Each slave will automatically forward requests and bring back the vote result to the software upgrader. Instead of the software upgrader, each slave waits for its module to complete transaction preparation. At last, after every slave passes its voting result to the software upgrader, the software upgrader will make a decision according to the outcome of voting. Because slaves and their target modules can concurrently execute commit or abort a command within their thread contexts, the performance of an upgrade transaction can be significantly improved when the software modules in an application are simultaneously executed on multiple processors.

The time spent in the upgrade transaction consists of two parts: *preparation time*, and *execution time of a transaction commitment* or *of a transaction abort*. Preparation time is the main part, which is the time for those replaced modules to reach their quiescent states.

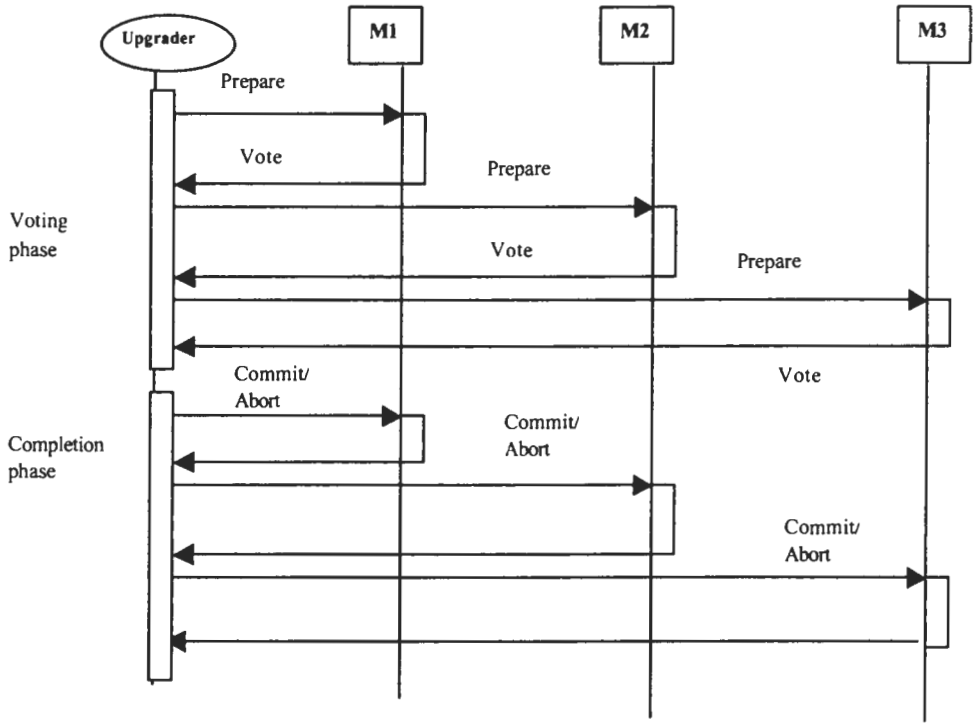


Figure 3.11. A sequential upgrade mode

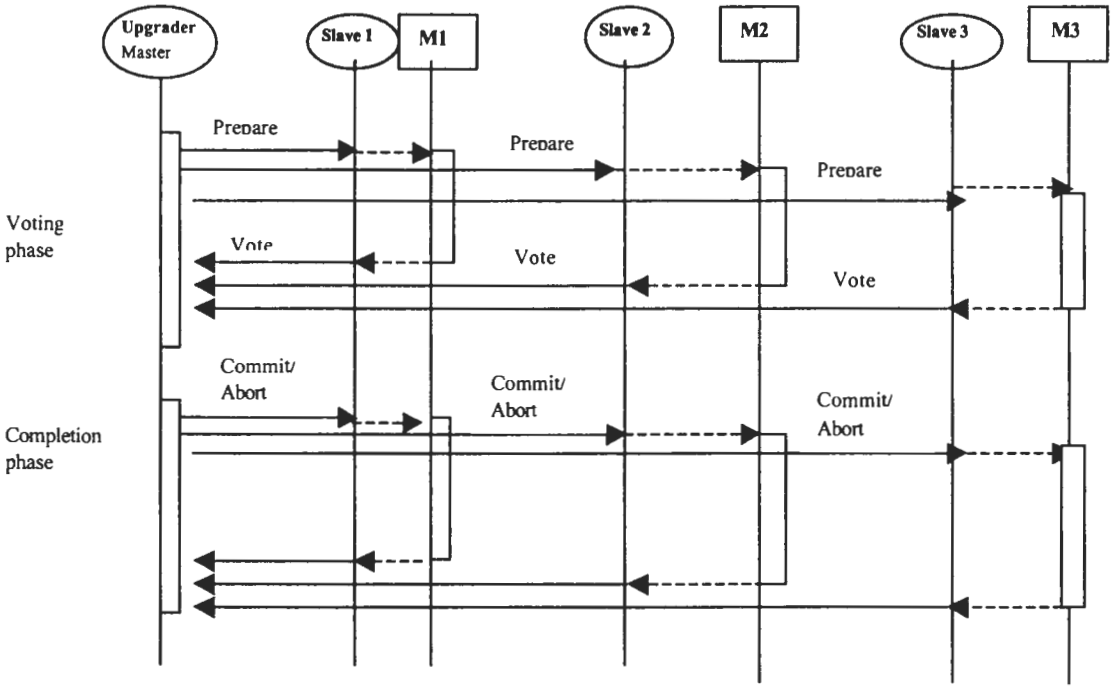


Figure 3.12. A concurrent upgrade mode

3.8 Summary

This chapter presented an approach to perform a live software upgrade addressing four areas: dynamic architecture and communication mode, reconfiguration management, run-time upgrade policy, and upgrade techniques. The **dynamic software upgrade framework** that we proposed can achieve the primary goals because of the following advantages:

1. Division of a module into *proxy* and *implementation* separates module communication from module computing. The *Name service* facilitates run-time discovery of modules. The *Publisher and Subscriber communication model* reduces overhead during run-time change. The dynamic architecture and communication model improve the flexibility and extensibility of software application.
2. Distinguishing between upgrading an individual and a group of modules in reconfiguration management provides great flexibility for the administrator to control the operation in application transition.
3. *Run-time upgrade policy* and *reaching quiescent state* can ensure application integrity when software modules are replaced, created or removed. Ability of adjusting *scope change* facilitates change management for an application administrator.
4. The *two-phase commit protocol* allows the on-line software change without influencing other modules and keeps property of "all or nothing" in all application transitions.
5. *The concurrent upgrading mode* minimizes the services downtime to improve service availability.
6. The upgrade transaction bounded with time constraints ascertains a *real-time upgrade*.

Chapter 4

Implementation

The design of the dynamic software upgrade framework was presented in the previous chapter. This chapter describes how to implement the framework, including strategies on the implementation of each component, queuing analysis, design patterns and concurrent control, state transition diagrams, class diagrams and object interfaces.

4.1 Implementation environment

We use an object-oriented methodology to implement the framework, since object orientation is not only a popular programming paradigm, but also a good vehicle for encapsulation and inheritance of software modules.

As discussed above, a live software upgrade really depends on the operating system and the programming language. Since our target is a multi-services application where multiple software modules are concurrently executed in their own thread contexts, a multi-threaded operating system is needed. Furthermore, dynamic loading and creation of modules should be supported by either the operating system or the programming language. Live software upgrades can be potentially applied in most safety- and mission-critical software applications, where C++ and C are the programming languages of choice. Java is a platform independent language, which has many attractive characteristics such as dynamic loading, creation of any class and run-time reflection. However, it is still not widely used in these software applications developed in industry today. As C and C++ do not support run-time loading and linking, the operating system needs to provide similar features instead. We require that those features be supported by the operating system.

To adapt to different environments and reduce the portability problem, ***ACE*** (Adaptive Communication Environment) [19][20][21] is introduced for the

implementation of our framework. ACE is an ideal environment since it is an object-oriented framework and toolkit that implements core concurrency and distribution patterns for communication software. It includes several components to help in the development of communication software and to achieve better flexibility, efficiency and portability. Some components such as concurrency and synchronization, memory management, timers, and thread management can be reused to build a live software upgrade framework. By using ACE, a dynamic software upgrade framework can be mapped onto many platforms such as most versions of Unix, Linux, and Win32 with little effort.

4.2 Module loading and creation

In this section, we present how to use dynamic link library and object-oriented methodology to load and create the module.

4.2.1 Dynamic link library

The dynamic link library (DLL) [22] [23] is a collection of small programs, any of which can be called when needed by an executing larger program that is running. This feature can augment code at run-time and defer the decision of bounding between the program and the particular library it references. It has already been utilized in Windows NT and Unix-like systems. Since each module needs to contain its own functions, states, variables, it can be separately compiled as a shared library. The software application can use the DLL facility to dynamically load a library into memory using the library name. Unix-like operating systems such as Solaris and Linux provide a family of routines such as *dlopen(), dlclose(), dlsym()* that allow a software application to directly access those uploaded shared libraries. A process can load a shared library at run-time using the *dlopen()* call, which instructs the run-time linker to load this library. Once the library is loaded, the application can call any function within that library using the *dlsysm()* call to

determine its address. Finally, the program can call *dlclose()* to unload the library from memory when the application no longer needs the shared library [24].

DLL API	DESCRIPTION
dlopen ()	Open a shared object. It makes a shared object available to a running process. dlopen () returns the process a "handle" which the process may use on subsequent calls to dlsym () and dlclose ().
dlsym ()	Get the address of a symbol in a shared object. It allows a process to obtain the address of a symbol defined within a shared object. Handle is either the values returned from a call to dlopen ().
dlclose ()	Close a shared object. It disassociates a shared object previously opened by dlopen() from the current process. Once an object has been closed using dlclose(), its symbols are no longer available to dlsym ().

Figure 4.1 Facility for *dynamic linking library*

4.2.2 Object-oriented methodology

If we say that a dynamic library facility allows an application at run-time to load a module library that has been complied as a shared library, the object-oriented programming style further allows the application to create an instance of a dynamic module at run-time. Object–oriented programming is often defined as a combination of abstract data types (ADTs) with inheritance and dynamic binding.

Inheritance: It partitions system architecture into sub-components which are related hierarchically. It allows a class to inherit the properties of another class or a set of classes. It separates, to some degree, the design from the implementation. It provides great ability to modify and reuse sections of the inheritance hierarchy without disturbing existing code by changing implementation of ancestors.

Dynamic Binding: The decision is made at run-time based upon the type of the actual object. It is more flexible since it enables developers to extend the behavior of an application transparently. The derived classes may be able to provide a different implementation. At object-invocation time, an appropriate implementation is selected at run-time.

In general, the combination of inheritance and polymorphism is a fundamental capability to form abstract data types such as base classes, and allows derived class to extend functionality from a base class, and to make appropriate run-time binding.

4.2.3 Constructing a dynamic module

In the design of a dynamic software upgrade framework, module-implementations are considered as dynamic modules, which can be modified and replaced in isolation, without affecting other existing components in an application. In order to utilize the features both from a dynamic link library and from an object-oriented methodology, dynamic modules should meet the following requirements:

- All the application-specific functions and state information is encapsulated in module-implementation components. Each module-implementation is wrapped as a C++ class, which is derived from a base class called *ModuleImp*.
- This class can be compiled separately as a dynamic shared library so that it can be loaded into and unloaded from memory at run-time.
- The base class *ModuleImp* contains some basic API functions used for manipulation of a live software upgrade. Furthermore, it should include some well-known and application specific interface, which can be overridden by its subclass. For example, function *serve ()* can be defined for message dispatching.

Because of using inheritance and late binding, the base class *ModuleImp* is known by the compiler at compile-time and its underlying implementation class can be changed at any point during run-time. All the public member functions in the base class have

already been exported to the application. The newly created instance of this subclass can be dynamically type cast to its base class *ModuleImp*. Therefore, at run time, it is possible to pull out an old version of the module-implementation and plug it into a new version of the module-implementation. All changes should be transparent to the whole application (see in Figure 4.2)

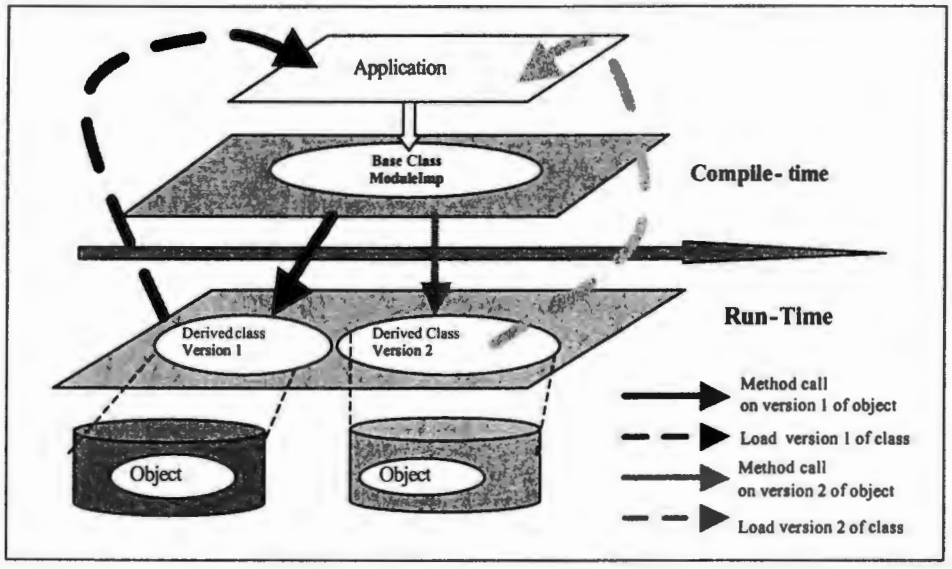


Figure 4.2 Dynamic invocations

4.2.4 Implementation of a module-proxy

A module-Proxy component is implemented by the Class *ModuleType*. A *ModuleType* object can be configured to delegate any *ModuleImp* object. Each *ModuleType* object contains a reference to a *ModuleImp* object with which it is currently bound. During an online change, the reference can be hot swapped to a new *ModuleImp* object.

4.3 The producer and consumer mechanism

Between a module-proxy and a module-implementation, there is a message queue. A module-proxy will receive all the incoming messages, and then put those messages into

this message queue. The producer and consumer mechanisms allow non-blocking message passing. Class *ModuleImp* has *putQueue()* and *getQueue()* methods. The *putQueue()* method enqueues the message while the *getQueue()* method dequeues the message. In order to introduce an asynchronous communication mode between modules, Class *ModuleType* also needs to have a *putQueue()* method, which is exported to other module objects (see Figure 4.3).

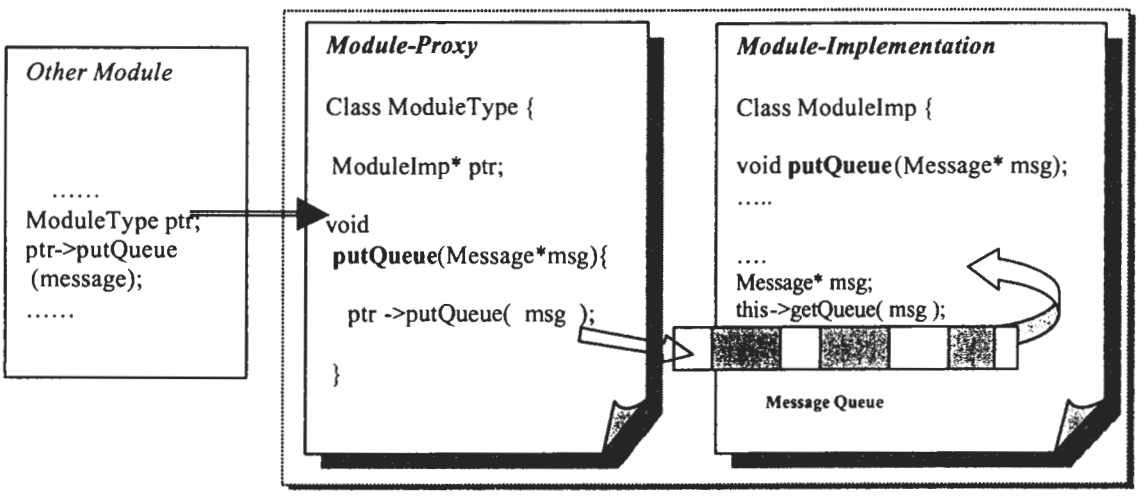


Figure 4.3 The producer and consumer mechanism

The common API function *service() can be* derived from the base class *ModuleImp* and be used for dispatching the message request into the corresponding instrumentation code (see Figure 4.4). The detailed processing functions have already been hidden from the public interface *service()*. The type of message is the key to passing the message into the appropriate function handler.

```
Int DerivedModuleImp::service () {
        For ( ; ; ) {
                ACE_Message_Block * msg = 0;
                getqueue( msg) {
```

```
                switch ( msg-> msg_type () ) {
                case ACE_Message_Block::MB_PROTO:
                        process_pdu ( msg -> base() );
                        break;
                case ACE_Message_Block::MB_DATA:
                        process_cli ( msg->base() );
                        break;
                case ACE_Message_Block::MB_PCSIG:
                        process_tic ( msg->base() );
                        break;
                }
                msg->release();
        }
        return 0;
}
```

Figure 4.4 Dispatching messages in function service()

4.4 Implementation of state consistency

Since the modules involved are not functional during a replacement transaction, the incoming service requests in the message format should be buffered for application consistency. A message-buffered queue is created by each module-proxy for this purpose. After an upgrade transaction, all the pending messages in the message-buffered queue should be processed by a new module-implementation. Figures 4.5, 4.6, and 4.7 give detail on how to achieve state consistency during an on-line module replacement.

Figure 4.5 shows that a module consists of a *ModuleType* object *proxy* and a *ModuleImp* object *impl_ver1*. Object *proxy* contains a reference *ptr* to its current working *ModuleImp* object, which is initialized with *imp_ver1*. When a module is signaled for

upgrade, a message-buffered queue is created by object *proxy* and further incoming messages are switched into the message-buffered queue. Afterwards, according to the run-time upgrade policy, the *proxy* object makes the *impl_ver1* object inactive. Then, it places a termination message at the end of the message-input queue.

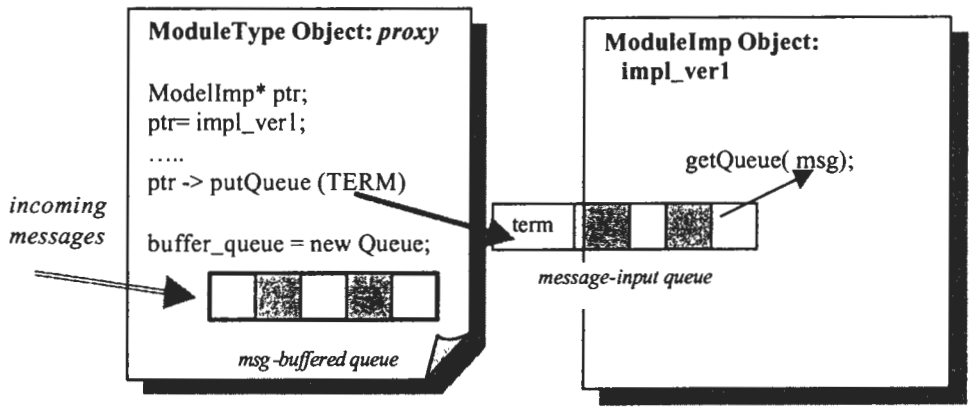


Figure 4.5 Sending a termination message

Figure 4.6 depicts that object *impl_ver1* continues processing the message until it gets the last TERM message. Since object *impl_ver1* assumes that there are no pending messages in its message-input queue after the TERM message, it finally terminates its service task and moves itself into a safe, so called the quiescent state. The object *proxy* begins to ask the object *impl_ver1* to capture a stable state and record it into a particular storage *state* by invoking the *putState* method. The object *proxy* is responsible for transferring the storage *state* to the new ModuleImp object *impl_ver2*. Likewise, the object *impl_ver2* is required to restore the state from storage *state* by invoking *getState* method. After finishing the state transfer, the object *proxy* hot swaps its reference *ptr*, which points to a new ModuleImp object *impl_ver2*. After that, the object *impl_ver2* becomes the current module-implementation. Finally, the message-buffered queue becomes the new message-input queue and is passed to the object impl_ver2.

Figure 4.7 illustrates that the object *proxy* begins to activate the object impl_ver2 and forward all the subsequent request messages to it. All the pending and incoming

messages placed into the message-input queue are processed by the object *impl_ver2*. Hence, there is no pending message being lost during the whole application transition. Meanwhile, the object *proxy* deletes the old object *impl_ver1*, and releases all the resources such as an old empty message-input queue. At last, everything is back to the normal.

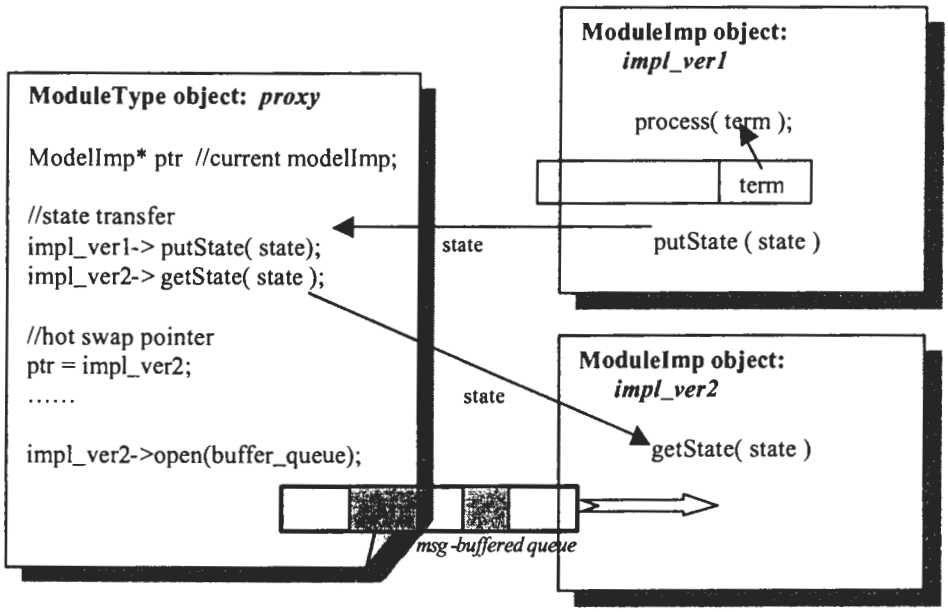


Figure 4.6 Hot swap of reference to the moduleImp object

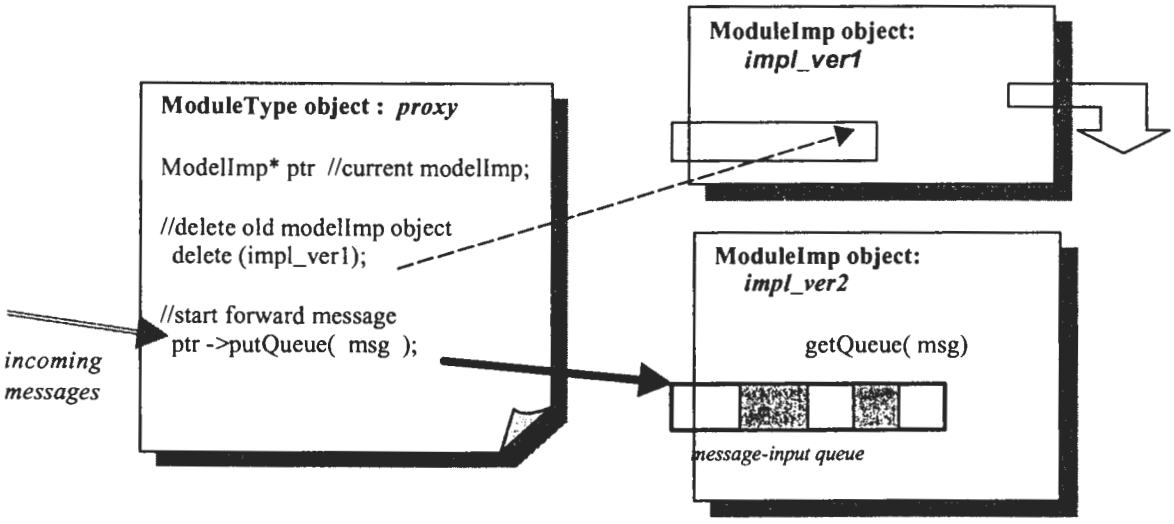


Figure 4.7 Removing the old module-implementation

4.5 Queuing Model

During the replacement of a module, queuing incoming request messages is an important strategy to achieve application integrity. Moreover, estimation of the size of a message-buffered queue is critical for the target module in order to avoid losing any pending messages.

A set of modules that send their messages to the target module should be identified first. In addition, the arrival rate of the messages from each module in the set should be known. For example, there are three modules P1, P2, P3, sending messages to Module *A* periodically. $\lambda_1, \lambda_2, \lambda_3$ are the sending rates generated by the set of modules {P1, P2, P3}. If the time taken by the target module for a live software upgrade is assumed to be $T_{transition}$, the size of the buffer queue should be at least $(\lambda_1 + \lambda_2 + \lambda_3) * T_{transition}$ (see Figure 4.8).

$$Q_{size} = \sum_{i=1}^{m} \lambda_i * T_{transaction}$$

Q_{size}: The minimum size of a message-buffered queue created by the target module

λ_i : The message arrival rate generated by module i.

$T_{transaction}$: The time taken by the target module for the live software upgrade.

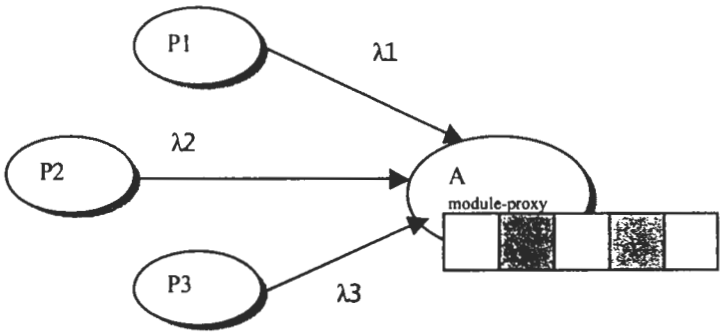


Figure 4.8 Queuing model

4.6 Design patterns and concurrency control

4.6.1 Factory pattern

To control the creation of dynamic modules fully, the **factory method** [25] can be utilized to delegate responsibility of creating the instances of the implementation version. The factory method defines an interface for creating an object, but lets its subclasses decide which class to instantiate. Every module library is forced to contain a factory method *create_module()*, which is used to create an instance. After loading a version of the module library, the application searches for the symbol of a function *create_module ()* to get the handle of the function. After the application invokes the factory method, the construction method of an implementation class will be further called to create an instance. Afterwards, the new object of the implementation class is dynamically cast to its base class *ModuleImp*. Finally, the reference to the new instance will be returned to the application. Therefore, the factory method gives derived classes a hook for providing an extended version of an object (Figure 4.9).

```
ModuleImp * create_module (void) {
        ModuleImp *moduleImp;
        /* create an instance of implementation class VRRPImp
          and type cast into its base class object.*/
        moduleImp = new VRRPImp();
        return moduleImp;
}
```

Figure 4.9 Factory method

4.6.2 Singleton pattern

It is very important for a dynamic software upgrade framework to have only one instance of a service component such as an event manager, a software upgrader or a name

service. A good solution is to use the **singleton pattern** [25], in which the class itself is responsible for keeping track of its sole instance and no other instance can be created, and hence can provide a well-known access point to access the instance. The Class *NameService, SystemManager, EventManger* implement the functionality accordingly in components: name service, software upgrader and event manger. The **singleton pattern** is applied in the construction of those objects.

4.6.3 Serialization of access to a critical regions

Many functions and variables in the event table and the name service are shared by modules in the framework. The access to critical sections needs to be serialized. *Acquire* access and *Release* access, as a mechanism for guaranteeing mutual exclusion, provides entry consistency. *Acquire* access is used to indicate that a critical region is about to be entered. *Release* access indicated that a critical region has just been exited. Acquire and Release should be put at the start and the end of each critical section, respectively.

The *Mutex* object can be put into critical regions in the *Name Service* and *Event Manager* classes. The module may enter a critical section by invoking the *acquire* method on a *Mutex* object. Any call to this method will block until the module that currently owns the lock has left its critical section. To leave a critical section, a module invokes the *release* method on the *Mutex* object it currently owns. ACE provides a class called *Guard* that ensures that a lock is automatically acquired and released upon entry and exit to a block of C ++ code, respectively [26].

As illustrated in the sample code of Figure 4.10, the Name Service class uses a *guard* object to define a critical region of code over which a lock is acquired and then released automatically when the block is entered and exited.

```
void NameService::insertModuleType(ModuleType* moduleType){
        //implicitly and automatically acquire and release the lock
        ACE_Guard<ACE_Thread_Mutex> guard(mutex_);
        //a critical section that adds new module context into name service.
        container.push_back( moduleType );
}
```

Figure 4.10 Serialization of access in Class NameService

4.6.4 Start Synchronization

When a group of modules is upgraded, it is very important to restart them simultaneously in order to maintain module dependency and state consistency. It is possible to use a **barrier,** which is a synchronization mechanism that prevents any modules from starting phase *n+1* of a program until all modules have finished phase *n*, to achieve the above objective. When a module arrives a barrier, it must wait until all the other modules get there too. When the last module arrives, all the waiting modules are resumed.

4.7 The Command Line Interface

The scripting language Tcl/Tk [27][28] provides a powerful capability to build application-specific commands into the Tcl/Tk kernel. It is easy to add new Tcl primitives by writing C procedures. An applications can define new Tcl commands. These commands are associated with a C or C++ procedure that the application provides. Thus, command line interfaces for dynamic application configuration are split into a set of built-in primitives written in a complied language and exported as Tcl commands.

The administrator could either run each command as a shell command, or group those commands into a configuration file and execute it as a batch file. Table 4.1

describes the functionality of each configuration command. Figure 4.11 illustrates the syntax rules for the construction of those commands.

COMMAND SYMBOL	DESCRIPTION
Enable	Enable the service of a module
Disable	Disable the service of a module
Load	Add a specified module
Unload	Remove a specified module
Replace	Replace a specified module
Syn_Replace	Simultaneously replace a group of modules
Syn_Load	Simultaneously load a group of modules
Syn_Upgrade	Simultaneously upgrade a group of modules. It not only replaces existing old modules but also adds new modules
SetTimer	Set timeout value for target modules to reach their quiescent state

Table 4.1 Command Line Interface Directives

<config-entires> ::= <enable > | < disable> | <replace> | <load> | <unload> |
<syn_replace> | <syn_load> | <syn_upgrade> |
<set_timer>

<enable> :: = Enable <module-name>

<disable> :: = Disable <module-name>

<load> ::= Load <module-name>

<unload> ::= Unload <module-name>

<replace> ::= Replace <module-name>

<syn_replace>::= Syn_Replace <module-parameter-list>

<syn_load> ::= Syn_Load <module-parameter-list>

<syn_upgrade> ::= Syn_Upgrade <module-parameter-list>

<set_timer> ::= SetTimer <value>

<module-parameter-list> ::= <module-parameter-list> <module-name>
<module-library-location> | NULL

<module-name> ::= STRING | NULL

<module-library-location> :: = STRING | NULL

<value> = STRING | NULL

Figure 4.11 EBNF format for Dynamic Configuration Entry

The set of configuration commands defines all the possible upgrading transactions, which should be fully controlled. For example, for updating a group of modules, when to update and how to start them simultaneously need to be totally ordered. For instance, the command *Syn_Upgrade* is designed for an application to replace the existing modules and add some new modules. It covers most functions in the command *Syn_Replace* and command *Syn_Load.*

Figure 4.12 depicts the sequence of a transaction that is defined in the command *Syn_Upgrade*. Through this transaction, two existing modules, *M1* and *M2*, should be updated and a new Module *M3* should be added into an application program. After the transaction, Modules *M1, M2*, and *M3* will be started at the same time. As barriers are introduced to synchronize the actions, it allows the upgrader to order the sequence. In general, a Module *M3* will be allowed to load into the application, after the upgrader gets a reply that all the replaced modules have been prepared for replacement. Then the upgrader requests that Modules *M1* and *M2* commit to replacement. Meanwhile, a module *M3* is added into the application. All the target modules are blocked waiting to start. Finally, the involved modules are started by the upgrader at the end of the

transaction. Likewise, the same strategy can be utilized in command *Syn_Replace* and command *Syn_Load* to synchronize the sequence in their transactions.

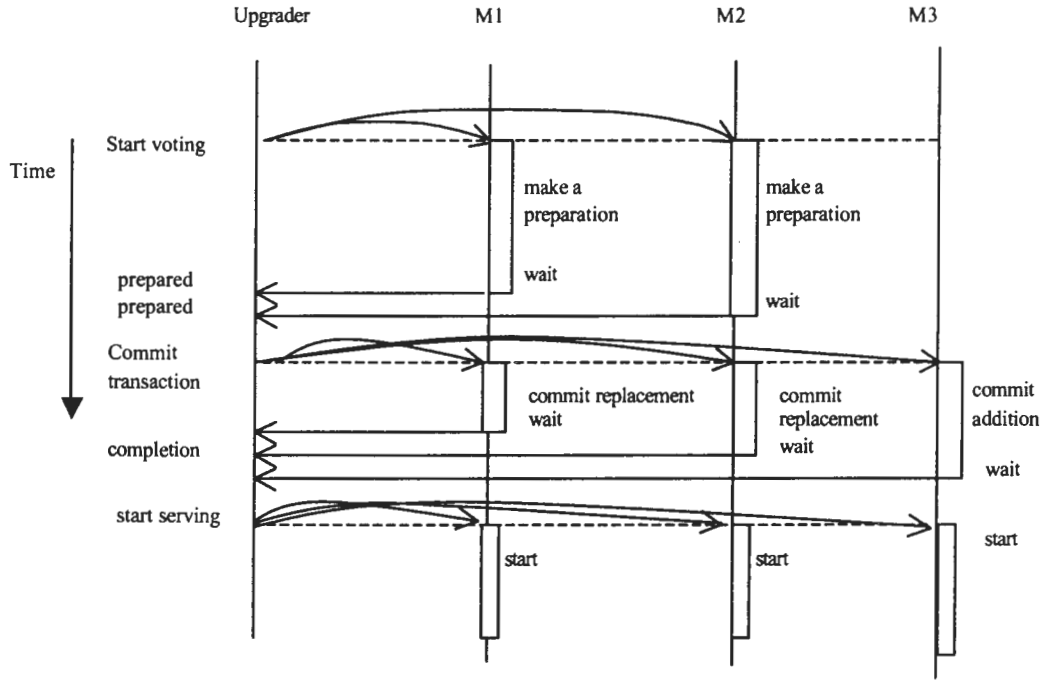


Figure 4.12 Synchronization in a transaction of Syn_Upgrade

4.8 State machine model

Among all of the possible upgrading transactions, the replacement of modules is more complicated than addition and removal of modules. In order to implement the two-phase upgrade protocol, it is very important to model the state transition of a software upgrader, a module-proxy and a module-implementation component during replacement of modules.

4.8.1 State machine of a software upgrader

When a software application starts, the software upgrader goes to the *Listening* state. Once it receives a command message from an administrator, it should look up the content in the command message and move to the *validation check* state.

Firstly, the software upgrader checks whether the command type is in the configuration command suite. Secondly, by looking up the name service, the software upgrader knows whether the replaced modules have been already loaded into the application or not. If a failure of passing the validation check occurs, the software upgrader then discards this command and returns to its *Listening* state. Otherwise, the software upgrader moves into the *Preparation* state.

Within the *Preparation* state, the software upgrader will load every new module library into memory according to the parameter list in the command message. After loading, it opens the library and searches for the symbol of creation function. By getting the handle to the function, the software upgrader creates one instance of replacing *module-implementation*. If the software upgrader can load library and create the instance of module successfully, it switches to the *voting* state. If there is anything wrong with those operations, it will go to *cleanup* state and do clean up.

At the *voting* stage, the software upgrader initiates the requests to require all the participating modules that are involved in this transaction. It then blocks and waits for the voting results from all module proxies. Next, every *module-proxy* begins to buffer all

incoming service requests and helps the *module-implementation* reach its *quiescent* state. If timeout for a *module-implementation* to reach the *quiescent* state occurs or it fails the *putState*, its *proxy* returns NO to the software upgrader.

If all of the returned results are YES, the final voting_result becomes PREPARED, the software upgrader moves to the *Commit* state, and invokes the commit method on all the participants. Finally, it updates information in the version control repository, and sends back a message to the administrator showing that the command was successfully committed.

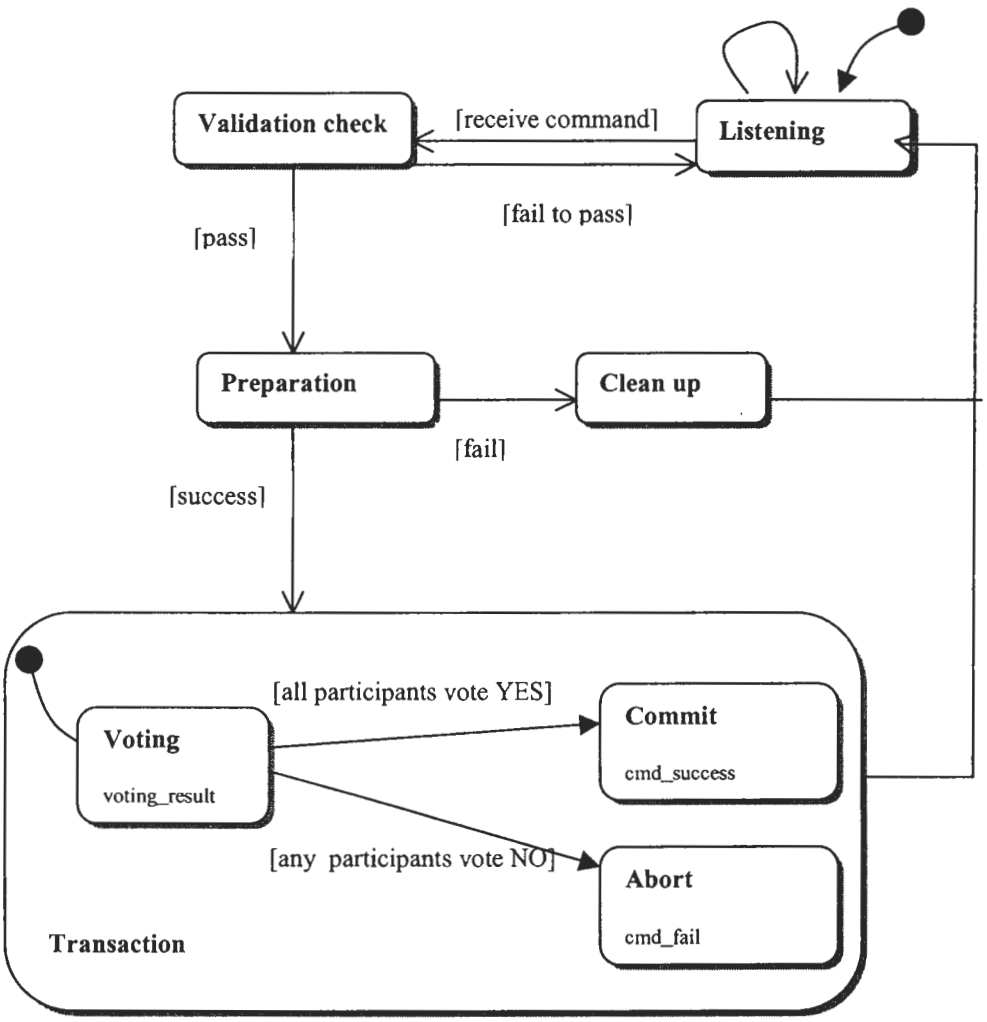


Figure 4.13 The state machine of the software upgrader

If any of the returned results are No, the final voting result becomes NO, and the software upgrader moves to the *Abort* state, thus aborting the whole transaction. A message showing that the transaction command failed will be sent back to the administrator (as shown in Figure 4.13).

4.8.2 State machine of module-implementation

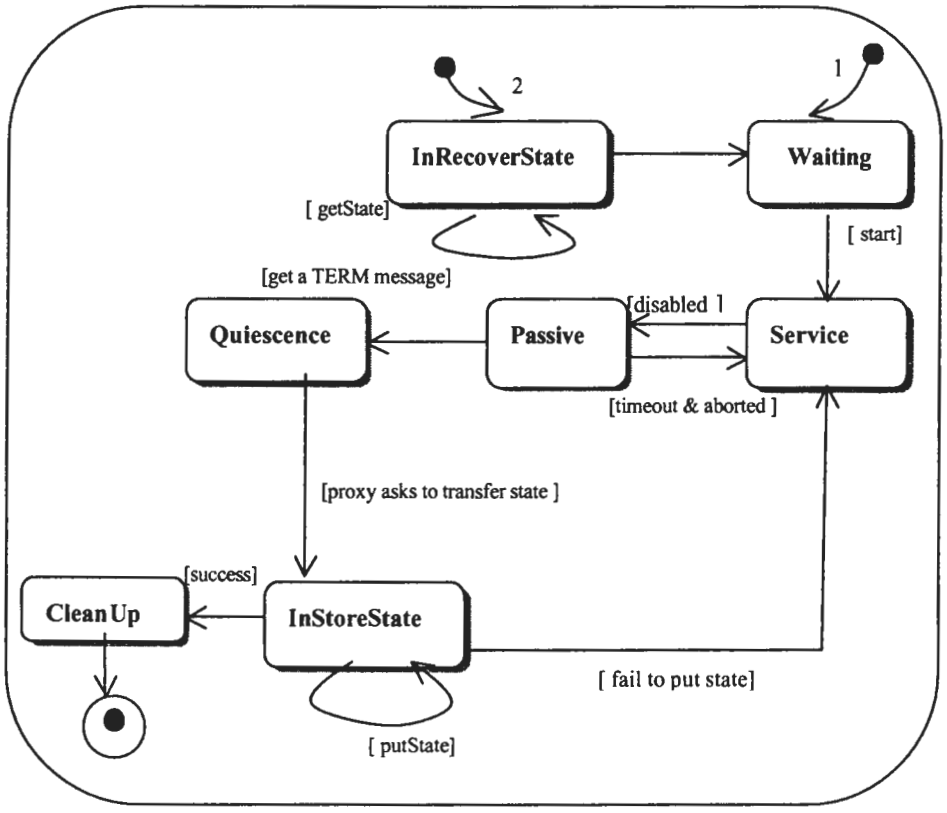


Figure 4.14 the state machine of module-implementation

Figure 4.14 shows that there are two entries for the module-implementation. The software upgrader normally loads a module library and creates one instance of module-implementation. After initialization, module-implementation will go through the first entry to the *waiting* state.

In the *Waiting* state, the module-implementation is blocked and waits for other module-implementations to finish their initialization. Finally, all the modules start together, and each goes into the *Service* state.

In the *Service* state, the module-implementation may not only serve incoming requests forwarded by its module-proxy but also initiate its requests to other modules.

When this module is involved in a replacement transaction, its proxy stops forwarding the incoming messages and begins to buffer them and send one termination message to its module-implementation. The module-implementation is disabled by its proxy and moves itself into *passive* state in which the module-implementation stops initiating its service requests to other modules.

During the *passive* state, it still serves incoming requests in its message queue until it gets a termination message from its proxy. Then the module-implementation terminates all service tasks. Finally, it informs its module-proxy that it has reached the *quiescent* state. If module-implementation still cannot reach quiescent state during timeout, it is forced to go back to Service state.

When proxy gets a reply, it asks its module-implementation to store its state. During the *InStoreState*, module-implementation puts the internal stable state into particular storage, which is passed to its proxy. If there is something wrong with extracting its internal state, it will be brought back to the *Service* state.

After storing its state, module-implementation moves itself into the *cleanup* state. Later, proxy removes this stale module-implementation from memory and releases all the hold resources as well.

The second entry shows that the *InRecoverState* reached after software upgrader has loaded new replacing module-implementation, and the module-proxy has transferred state to the new module-implementation from old one. In the *RecoverState,* the replacing module-implementation gets an opportunity to be consistent with the replaced module. Then it switches into the W*ait* state and is ready to start.

4.8.3 State machine of module-proxy

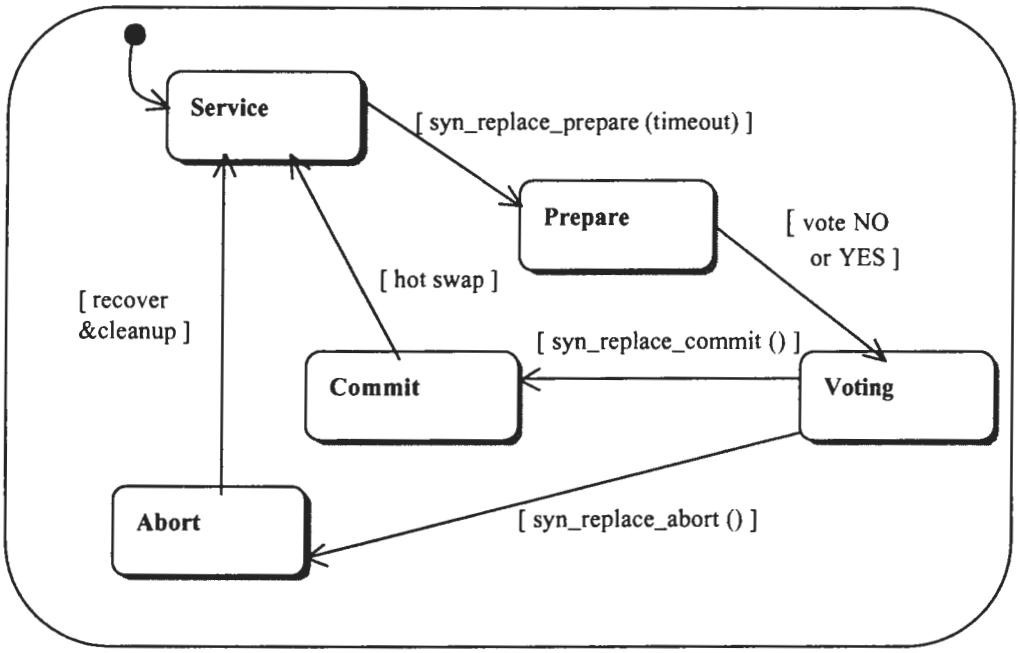


Figure 4.15 the state machine of module-proxy

As shown in Figure 4.15, module-proxy coordinates with software upgrader to synchronously upgrade the module-implementation. In the beginning, module-proxy is in the *Service* state where it puts all the incoming messages into the message queue.

When the software upgrader requires all the participating module-proxys to be involved in a transaction such as replacement, it invokes *syn_replace_prepare ()* on all the proxys and requires that each module make a preparation in a given time. During the *Prepare* state, each module-proxy intercepts all the incoming messages and puts them into a new message queue. It makes the module-implementation inactive and sends it a termination message. After that, it becomes blocked waiting for proxy-implementation to reach the quiescent state. If a timeout occurs, it will return the vote "NO" to software upgrader. If module-implementation reaches the quiescent state within a given time value, module-proxy instructs its module-implementation to put the internal state into

particular storage. If module-implementation successfully extracts the state, module-proxy will return the vote "YES" to the software upgrader. Otherwise, the vote "No" will be returned to the software upgrader.

All the involved module-proxys need to vote. If all module-proxy votes are "YES", software upgrader will announce that transaction should be committed. Then syn_replace_commit method is invoked on each involved module-proxy.

During the *Commit* state, module-proxy transfers the state from old replaced module-implementation to the new one, and asks the new one to recover the state. Additionally, it passes the buffered message queue to the new one. Then it hot-swaps its reference point to the new module-implementation. Finally, it executes a destruction method on the stale module-implementation, unloads it from memory, and makes the new module-implementation active.

If any of the module-proxies vote ABORTED, the software upgrader will ask all the module-proxies to abort the transaction. In the *Abort* state, the proxy passes the buffered message queue to the old module-implementation and brings back old module-implementation into the *Service* state. Meanwhile, it removes the new module-implementation from memory.

After the transaction is committed or aborted, the new module-implementation or the old one is active. Finally, the transaction is completed and the module-proxy goes back into the original *Service* state.

4.9 Main class diagram

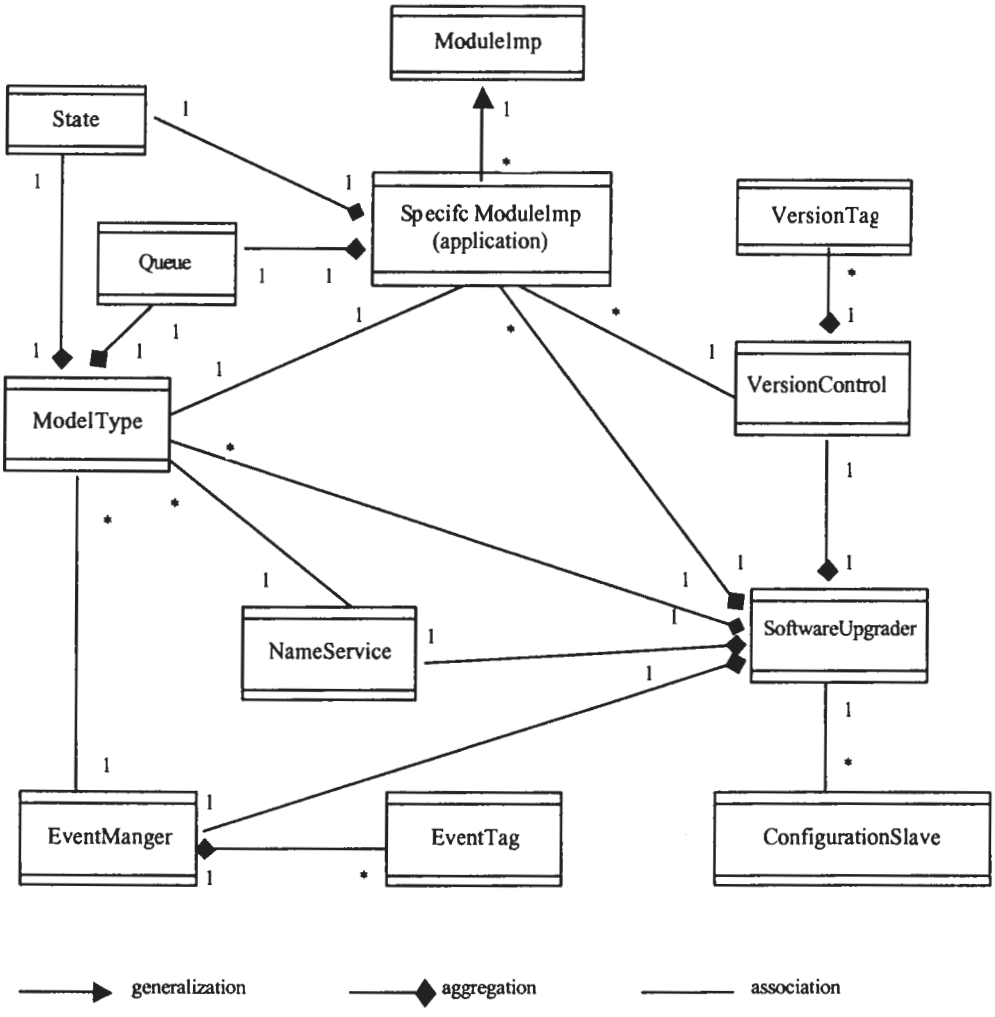


Figure 4.16 Main class diagram

4.10 Object Interface

METHOD	DESCRIPTION
getName	Returns the name of module when querying
setModuleImp	Associates with one ModuleImp object and delegates it
putQueue	Used by other modules to send request messages and forward messages to its ModuleImp object.
EnableModule	Handles an enable request from software upgrader and activates ModuleImp object
DisableModule	Handles a disable request from software upgrader and make ModuleImp object inactive
TerminateModule	Sends a termination message to its ModuleImp object
loadModuleImp	Handles a loading request from software upgrader and links it with implementation object and creates one message queue for implementation object.
syn_replace_prepare	When gets prepare request from software upgrader, it helps its ModuleImp object reach quiescent state, and retrieves the state.
syn_replace_commit	Handles a commit request from software upgrader. It replaces its ModuleImp object, remove the old one.
syn_replace_abort	Handles an abort request from software upgrader. It aborts its part of transaction and recovers the original status.

Table 4.2 *ModuleType* Interface

METHOD	DESCRIPTION
open	Starts a task, given with reference to input message queue, name service, event manager
service	Implements all the application specific functions. It is called by open () and overridden by derived class.
terminate	Automatically called when module finish service ()
enable	Activates the task. It should be overridden by derived class and called by resume ()
disable	Makes the task inacitve. It should be overridden by derived class and called by suspend ()
suspend	Handles the suspend request from proxy. It then call disable ()
resume	Handles the resume request from proxy. It then call enable ()
initialize	Implements initialization for the task. It is overridden by derived class and called by open ()
send_termMsg	Implements termination for the task. It appends a termination request message at the end of the message queue. It is called by ModuleType object
put_state	Handles the request for capturing a state. It is called by a ModuleType object
get_state	Handles the request for recover the state. It is called by a ModuleType object
abortTransaction	Handles the request for abort transaction request. It is called by a ModuleType object
putQueue	Puts a request message into queue. It is used by a ModuleType object to forward messages
getQueue	Blocks until receiving a message from queue

Table 4.3 *ModuleImp* Interface

METHOD	DESCRIPTION
getNameService	Implements singleton pattern, and returns only one ref to name service object.
setModuleUsageStatus	Used by software upgrader to set usage status of a module
findModuleStatus	Used by software upgrader to find out the usage status of the module.
insertModuleType	Used by software upgrader to register new ModuleType object when loading a new module.
findModuleType	Resolves the reference to module by a module's name
removeModuleType	Unregisters the module type when unloading a module.

Table 4.4 *NameService* Interface

METHOD	DESCRIPTION
getVersionControl	Implements singleton pattern, and returns only one ref to VersionControl object.
insertVersionTag	Used by software upgrader to register new module library
findVersionTag	Used by software upgrader to resolve version tag by a module's name.
removeVersionTag	Used by software upgrader to unregister module library, and remove it from memory

Table 4.5 *VersionControl* Interface

METHOD	DESCRIPTION
getSoftwareUpgrader	Implements singleton pattern and returns only a softwareUpgrader object.
loadLibrary	Opens the module library and searches the symbol of a function used for creation of a dynamic module
closeDll	Closes the module library
make_connection	Sets up listening port
voted_by_slave	Used by subordinate slave to vote
voting_result	After sending prepare command to modules involved, it blocks and waits for final voting results
enable_module	Handles a request that enables module from CLI.
disable_module	Handles a request that disables module form CLI (command line interface)
Load_module	Handles a request that loads a module from CLI
unload_module	Handles a request that unloads a module from CLI
replace_module	Handles a request that replaces a module from CLI
Syn_load_module	Handles a request that synchronously loads a group of modules
Syn_replace_module	Handles a request that synchronously replaces a group of modules
Syn_upgrade_module	Handles a request that synchronously upgrades a group of modules
set_timer_value	Handles a request that reset the time value for upgrade transition.
process_command	Gets requests from CLI and dispatch requests.

Table 4.6 *SoftwareUpgrader* Interface

METHOD	DESCRIPTION
getEventManager	Implements singleton pattern and returns only one Ref. to EventManager object
publishEvent	Used by publisher to publish its Event
clearEvent	Used by publisher to unregister its published Events
resolveEventTag	Used by subscriber to resolve event Tag by name of publisher and event type
subscribeEvent	Used by subscriber to subscribe event
unSubscribeEvent	Used by subscriber to unsubscribe event
notifyAll	Used by publisher to multicast published events

Table 4.7 *EventManager* Interface

Chapter 5

Simulation and Results

Chapters 3 and 4 describe the design and implementation of a dynamic software framework. This chapter provides the results of a simulation of the framework.

5.1 The simulation scenario

In order to demonstrate the dynamic software upgrade framework, an application was developed using the components and communication model described in Chapters 2 and 3. Both the goal of the demonstration and the architecture used for the demonstration application are described as follows.

5.1.1 Objectives for the simulation

1. *To demonstrate safe, reliable software upgrade in a non-stop application*

   This is the most important feature in the dynamic software upgrade framework. In a non-stop application, only target modules are affected in an upgrade transaction, while other parts continue functioning. Module dependencies should be shown in the simulation.

2. *To show the property of an atomic upgrade transaction: all or nothing*

   A two-phase commit protocol discussed in the framework is utilized for the implementation of an atomic upgrade transaction.

3. *To apply a concurrent upgrade mode*

   The simulation should use the concurrent upgrade mode to minimize the downtime of services in the application.

4. *To illustrate most of the transactions defined in the command line interface*

A variety of upgrade operations should be shown in the simulation. An individual module or a group of modules should be involved in those operations.

5. *To depict the ability to maintain application integrity*
6. *To demonstrate the run-time evolvable architecture and the ability to support incremental change*

   The simulation program should use components and strategies mentioned in the framework such as name service, event manager and version control repository. The publisher and subscriber communication model should be utilized in the application as well. The modules in the application can be added, removed or replaced incrementally and gracefully.
7. *To show the timing constraints in an upgrade transaction*

   The upgrade transaction in an application is bounded by a given period of time in order to reconfigure the software application in real-time.

5.1.2 Architecture of the experimental application

The demonstration application is called *non-stop Router*, which provides multiple services (as depicted in Figure 5.1). On the one hand, it periodically broadcasts ICMP (Internet Control Message Protocol) packets, HELLO discovery packets, and HEARTBEAT packets. On the other hand, there are three nodes, *IP monitor*, *OSPF monitor,* and *VRRP monitor*, which receive the ICMP packets, the HELLO packets and the HEARTBEAT packets accordingly. An application administrator can reconfigure the software application through the *CLI* (command line interface) node. New versions of the software can be transferred to the local machine by FTP. Thus, a live software upgrades can be implemented locally.

Figure 5.2 shows the architecture of the *non-stop Router*, which consists of four modules: *TIMER*, *IP*, *OSPF*, and *VRRP*, which are concurrently executed. To illustrate a publisher and subscriber communication model, the *TIMER* module that is initiated first,

acts as a publisher and publishes an event "Broadcast". Other modules will subscribe to the event "Broadcast" once they are alive. After a given period of time, the event happens. Hence, the *TIMER* module multicasts the event to the subscribers in the message format. In this scenario, the *TIMER* module does not know how many subscribers there are when the event occurs, meanwhile the subscribers can be removed and added from the list of subscription at run time.

Modules VRRP, IP and OSPF are all active modules, which not only serve incoming requests but also initiate requests. The *TIMER* module acts as an engine that stimulates the action of the three modules by sending event messages to those subscribers periodically. When the *IP* module receives an event message, it broadcasts an ICMP packet to the neighbor and meanwhile sends a message to the *OSPF* module. Similarly, g an event message is received periodically, the *VRRP* module not only broadcasts a HEARTBEAT packet to other nodes but also sends a message to the *OSPF* module. The *OSPF* module differs in behavior from IP and VRRP modules in that the *OSPF* module only broadcasts its HELLO packet after receiving an event message. In this scenario, the three modules in the non-stop Router not only communicate with external nodes but also communicate with each other concurrently.

Figure 5.3 shows that the packets sent to the nodes by a module are labeled with the timestamp and the version of the implementation. The three monitor nodes detect the health of the modules running in the router. If a module is in service and keeps broadcasting, its respective monitor will obtain the timestamp and the version of the sending module from any receiving packet. If the module is out of service, the monitor reflects the situation and waits to receive the packets that have been broadcast. Modules are out of service if the modules are disabled or currently involved in an upgrade transition.

In order to reflect consistency of an application before and after a live software upgrade, the content if the messages sent by the *VRRP* module and the *IP* module to the

OSPF module is an integer number (called number relay) in a consecutive format. In the *VRRP* module or the *IP* module, the integer number in the messages should be initialized to one. Afterwards, the number should be incremented for the next send. On the other side, the *OSPF* module will lookup the messages sent by the *VRRP* module or the *IP* module, and output the content in the messages. During a replacement transaction, the relay number, as the state in the old module-implementation, should be transferred into the new one, so that the new version of the implementation can synchronize its relay number with the old one. Thus, before, during or after the replacement of the module, the number in the content of messages sent by the *IP* module or the *VRRP* module should always be consecutive.

During replacement of the target modules, the subsequent messages will be buffered. And the new module-implementation will process those buffered messages after the transaction. Thus, no incoming messages are lost during a replacement transaction.

In order to show the module dependency, the message protocol used in Version 1 and Version 2 are different from the one in Version 3. If Version 1 and Version 2 of the modules are still running, Version 3, as a new loaded component, needs to accommodate both the old and the new message formats at run time in order to achieve backward compatibility.

5.2 Study of transactions

The non-stop application running in the router can be started, transformed and evolved incrementally by a series of configuration commands that form six consecutive upgrade transactions. These transactions are utilized to reflect the features of the framework from different viewpoints. Transaction 1 shows the module addition protocol and publisher-subscriber communication model. Transactions 2 and 3 show the module replacement protocol and state consistency respectively. They demonstrate how the

involved modules reach the quiescent state, synchronize the state, buffer messages and commit switchover. Transaction 4 shows the concurrent upgrade mode, the two-phase commit protocol and the synchronization of start. Transaction 5 depicts safe upgrade in case of failure. Transaction 6 illustrates the scenario of module dependency.

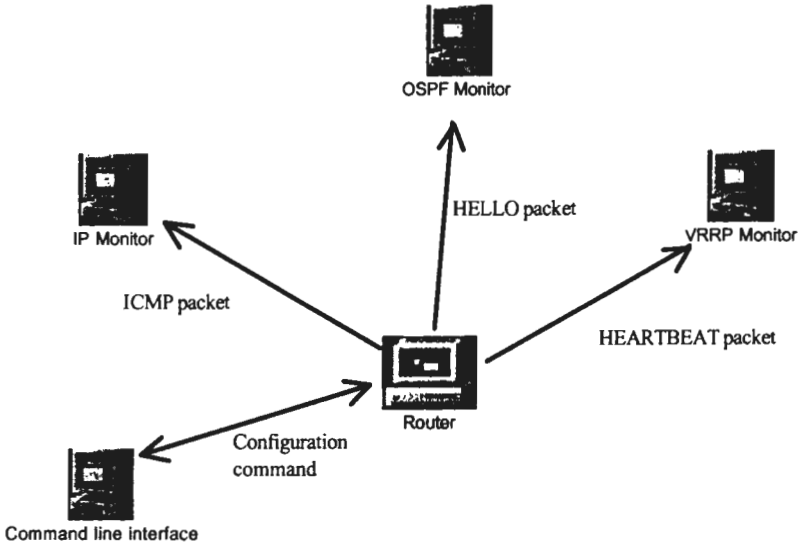


Figure 5.1 The structure of nodes

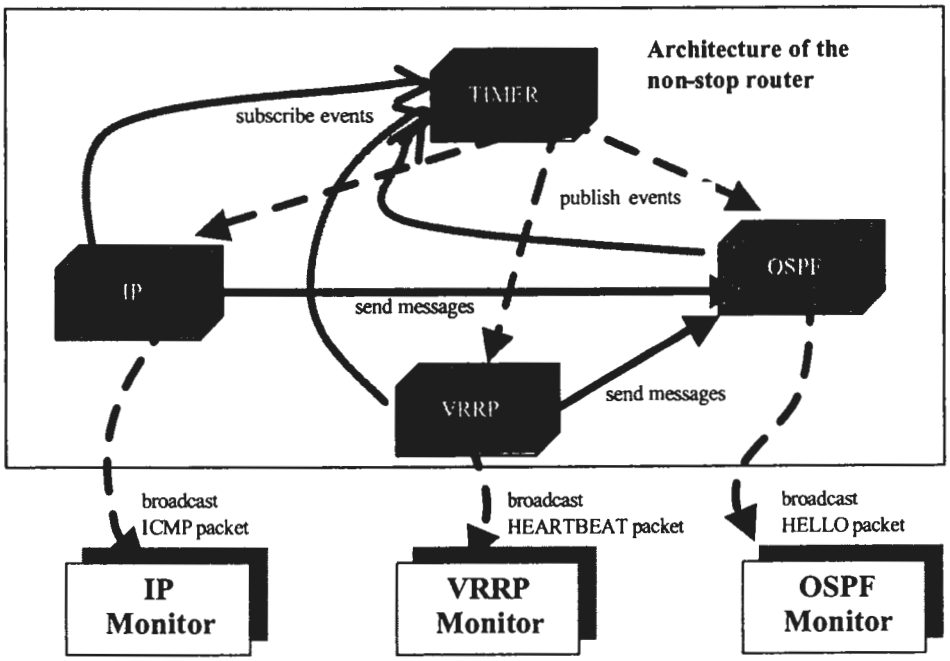


Figure 5.2 Architecture of the non-stop router

Figure 5.3 Screen output of a non-stop application

5.2.1 Transaction 1

Configuration Command

Load TIMER libTimerImp.so

Syn_Load OSPF libOSPFImp1.so IP libIPImp1.so

Figure 5.4 shows that at the beginning, the *TIMER* module is loaded by the administrator. Then the *TIMER* module publishes an event "broadcast" initially. After getting started, it periodically sends event messages to all the subscribers when event "broadcast" occurs in a time interval.

By execution of the second command line, the library of the *OSPF* module and the *IP* module are uploaded into the application. The instances of modules are created. In order to synchronize, the created modules are blocked waiting for completion of all the creation and initialization. Finally, the involved modules are permitted to start simultaneously by the software upgrader.

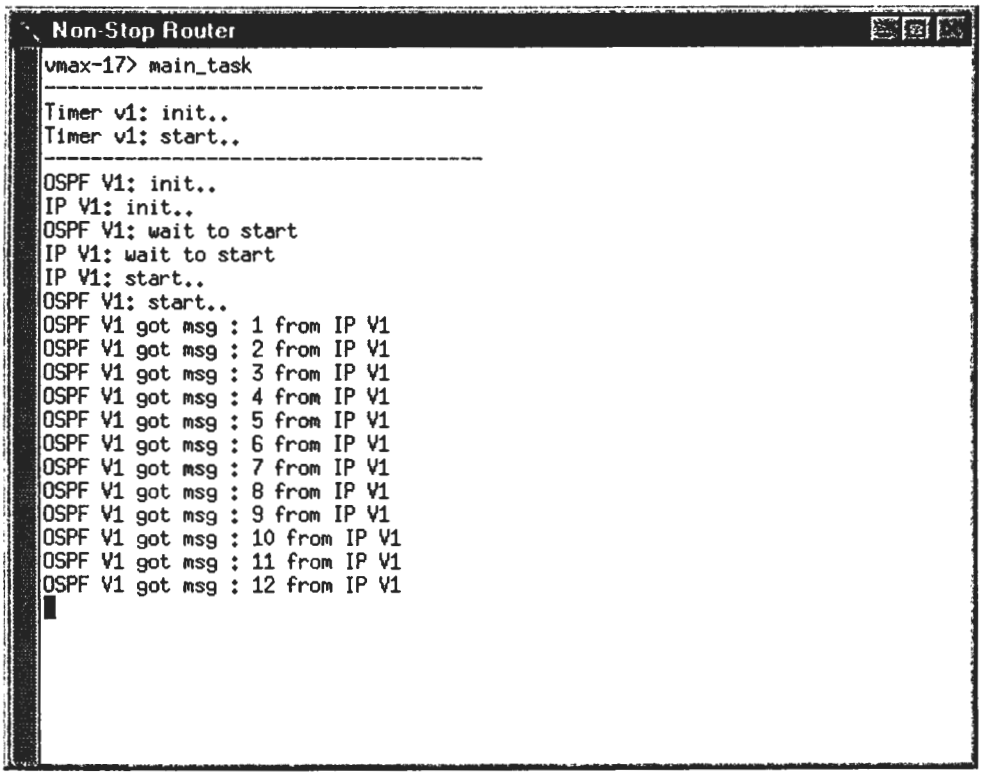


Figure 5.4 Transaction 1

After becoming active, the *IP* module and the *OSPF* module subscribe to the event "broadcast". As a result, the *IP* module and the *OSPF* module begin to broadcast

the ICMP packets and the HELLO packets to their neighborhood. By receiving event messages, the *IP* module is reminded to send messages to the *OSPF* module. The *IP* module can discover the address of the *OSPF* module by looking up the name service. Figure 5.4 indicates that the relay number in the content of the messages sent by the *IP* module starts form one and is incremented for subsequent sendings.

5.2.2 Transaction 2

Configuration command

Replace IP libIPImp2.so

Transaction 2 shows that Version 1 of the *IP* module-implementation can be successfully replaced with Version 2. In order to reach the quiescent state, the *IP* module-implementation is disabled by its module-proxy so that it stops broadcasting packets to the nodes and sending the messages to the *OSPF* module. After consuming all the pending messages in its input message queue, the module-implementation finally gets a special TERM message appended by its proxy. As a result, it terminates its task thread and reaches the stable state. Once the old implementation object has checkpointed its state and put its relay number into state storage, the new implementation object will get a chance to recover the state transferred by its proxy. In this case, Version 2 of the implementation object, *which* has the same state as Version 1, is ready to send its consecutive relay number in the messages again.

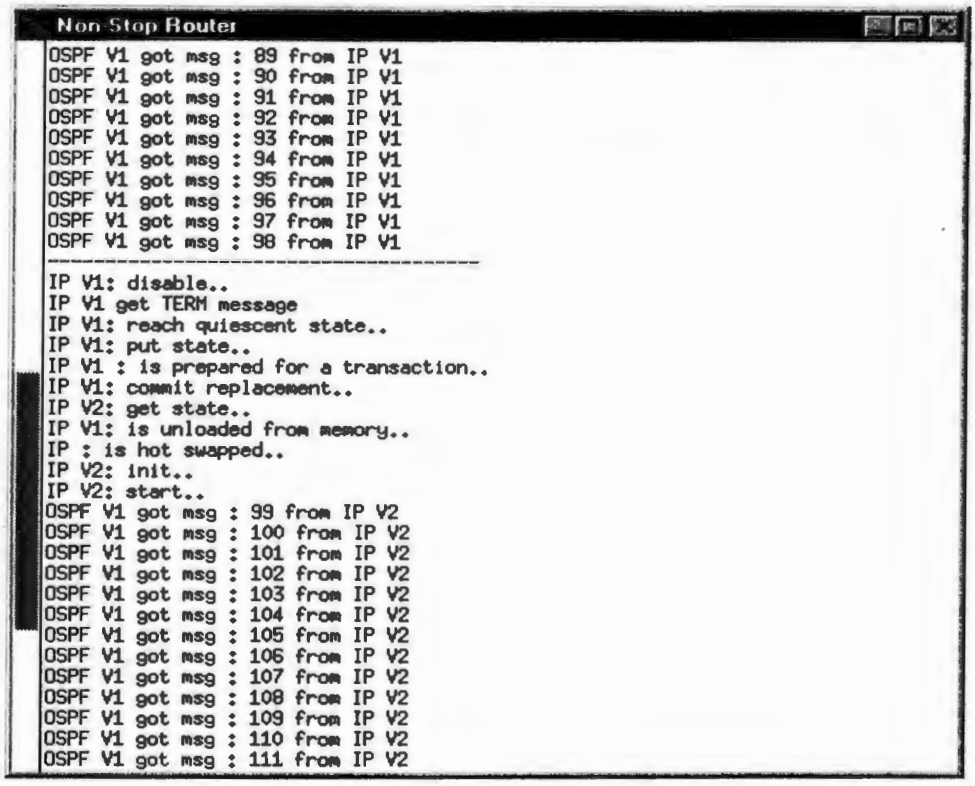


Figure 5.4 Transaction 2

As discussed in the module replacement protocol, after the module-proxy removes the old implementation object from memory and associates itself with the new one, the *IP* module is placed back into service status. Thus, the *OSPF* module will receive the messages from Version 2 of the *IP* module. The relay number in the messages keeps incrementing (see Figure 5.4).

5.2.3 Transaction 3

Configuration Command

Replace OSPF libOSPFImp2.so

Figure 5.5 demonstrates that during the replacement of the *OSPF* module, all the incoming messages are buffered into a message-buffered queue. After Version 1 of the

implementation object has been replaced by Version 2, the message-buffer queue is passed to the new implementation object to process. All the pending messages during the transaction and new incoming messages from the *IP* module are not lost.

Transactions 2 and 3 depict that transferring the state between two versions inside the target module and buffering the incoming requests from outside during a transaction can achieve state consistency.

Figure 5.5 Transaction 3

5.2.4 Transaction 4

Configuration Command

Syn_Upgrade OSPF libOSPFImp1.so IP libIPImp1.so

VRRP libVRRPImp1.so

Figure 5.6 demonstrates that the existing *OSPF and IP modules* are both replaced with Version 1 while a new *VRRP* module is added into the application simultaneously. The concurrent upgrade mode and the two-phase commit protocol have been applied in the transaction. Two existing *OSPF* and *IP* modules are required to update their module-implementations. According to the concurrent upgrade mode and the two-phase commit protocol, two slaves created by the software upgrader are in charge of the two replaced modules accordingly. At first, each replaced module is required to move into a quiescent state. Secondly, after the *OSPF* module and the *IP* module return their vote, the software upgrader decides the fate of the transaction based upon a final voting result. Once replaced modules have successfully been prepared, the software upgrader will commit the upgrade transaction.

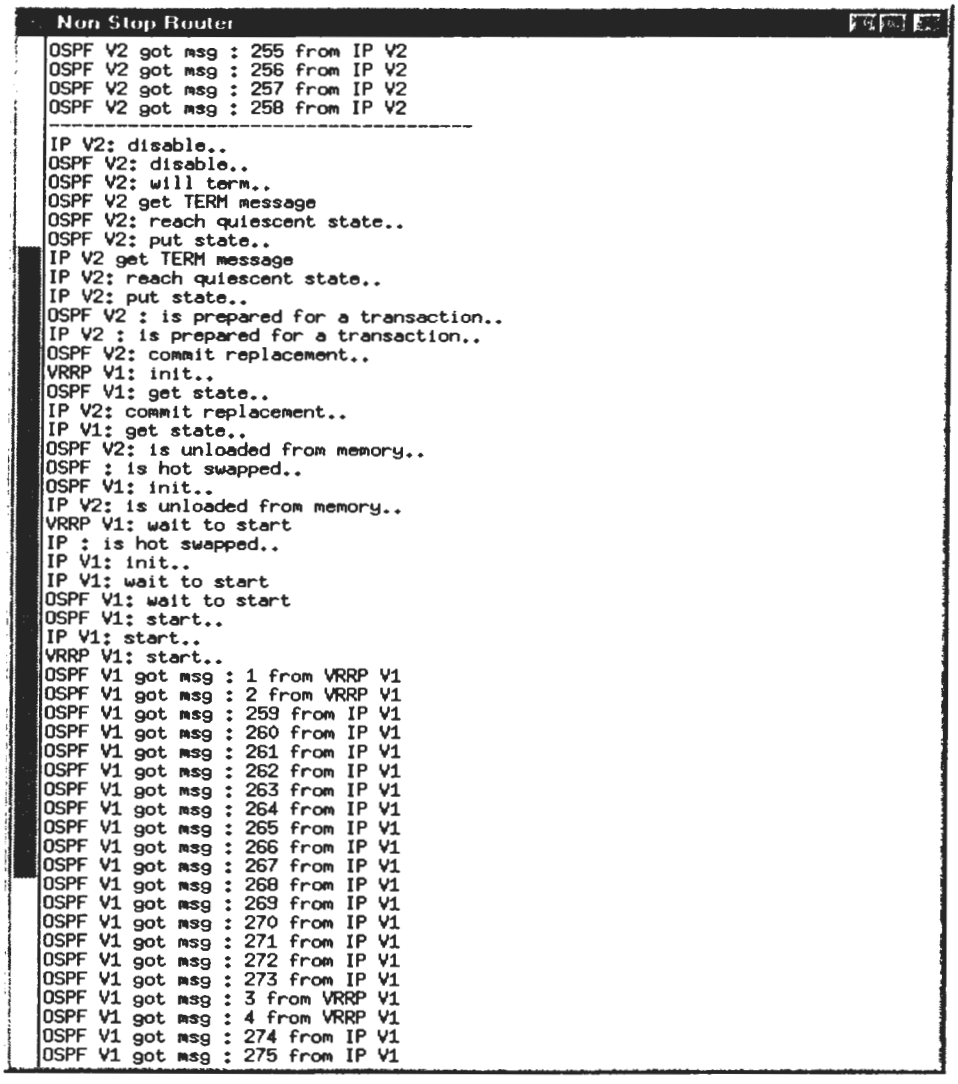


Figure 5.6 Transaction 4

Hence, the *OSPF* module and the *IP* module are replaced by their new implementations. Meanwhile, the *VRRP* module is allowed to upload, and one instance is created. Next, all involved modules are blocked waiting for start. Finally, the software upgrader initiates the start for all the modules. As shown in Figure 5.6, the states of the application are kept consistent after replacement of the *OSPF* module and the *IP* module.

5.2.5 Transaction 5

Configuration Command

SetTimer 2

Syn_Replace OSPF libOSPFImp2.so IP libIPImp2.so

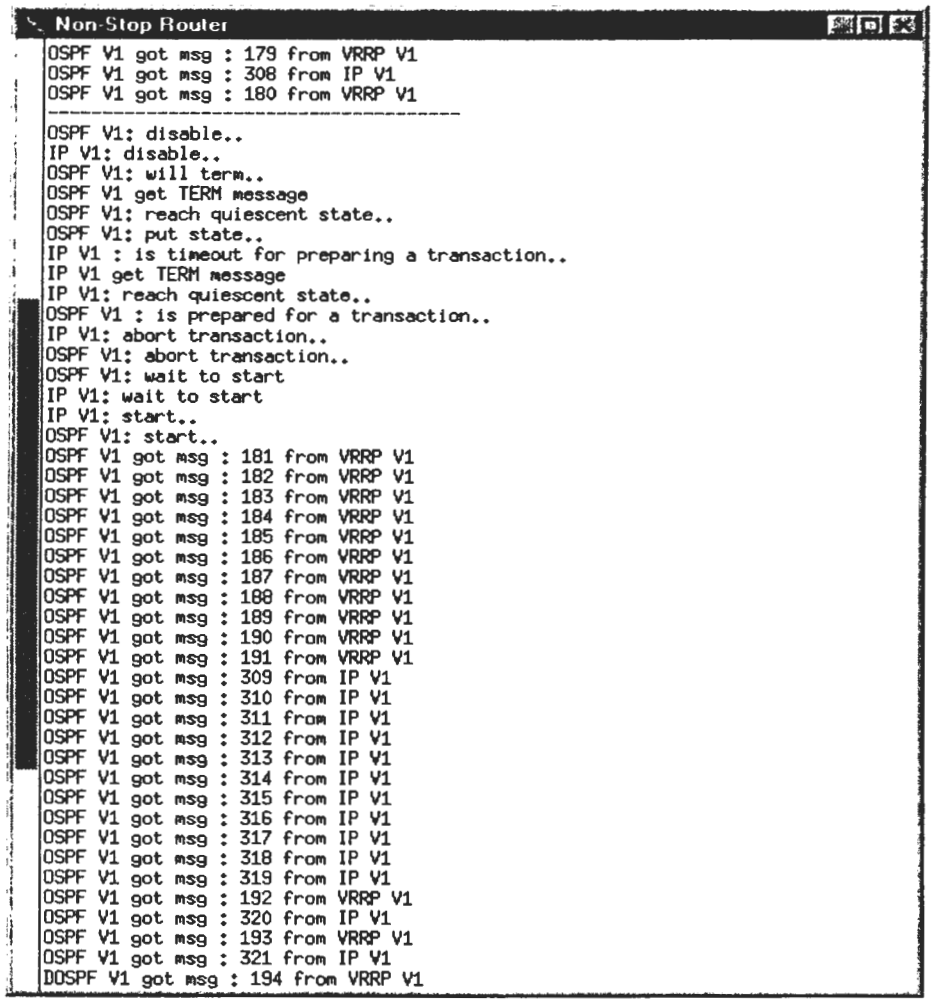


Figure 5.7 Transaction 5

Figure 5.7 depicts the scenario of aborting a transaction. The maximum preparation time is given as two seconds, which may be shorter than the time taken by one of the modules to make a preparation for a live upgrade. Within two seconds, the *OSPF* module can reach its stable state, return its vote and move into the *voting* state.

However, the *IP* module cannot meet the timing constraint. Thus, the *IP* module returns the vote "NO" to the software upgrader. The software upgrader decides to abort the transaction because one of target modules votes "NO". Consequently, the newly created implementation objects are removed from memory, and the original implementation objects are automatically brought back to service. Finally, the message-buffered queue is given to the current implementation object. Although the transaction is aborted, no pending messages or new incoming messages are lost.

5.2.6 Transaction 6

Configuration Command

SetTimer 10

Syn_Replace OSPF libOSPFImp3.so IP libIPImp3.so

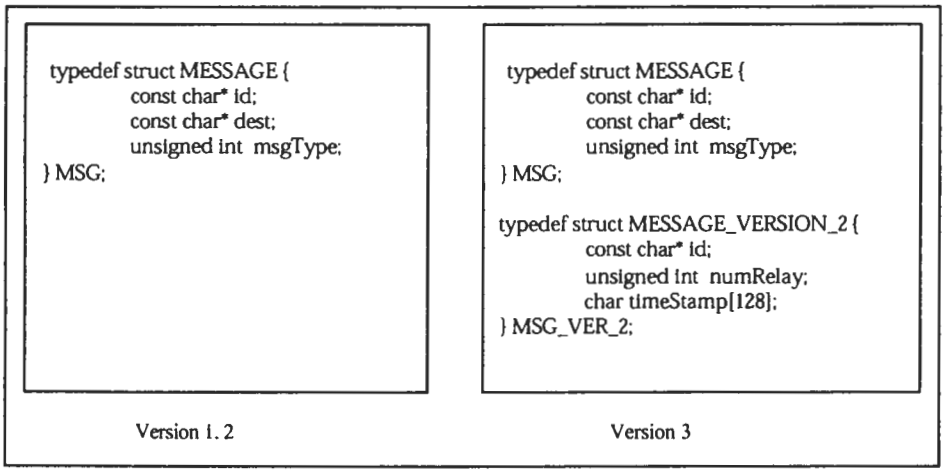


Figure 5.8 Message protocol

Figure 5.8 shows the message protocols used by different versions of modules *IP*, *VRRP* and *OSPF* for their communication. The message protocol used in version 1 and 2 of modules is shown on the left side of the figure, while the message protocol used in version 3 is illustrated on the right side. Version 3 of the message protocol adds a field of a timestamp to each message. In order to stay backwards compatible, version 3 of the the

the modules not only supports their new message format, but also accommodates the old one, such as Versions 1 and 2. All the messages are labeled with the versioning message protocol.

Figure 5.9 illustrates that after replacing version 1 of the *OSPF* module and the *IP* module with their Version 3 corresponding module, Version 3 of the *OSPF* module can either interpret the messages in the format of Version 1 sent by the *VRRP* module, or it can understand the messages in the format of Version 3 sent by the *IP* module.

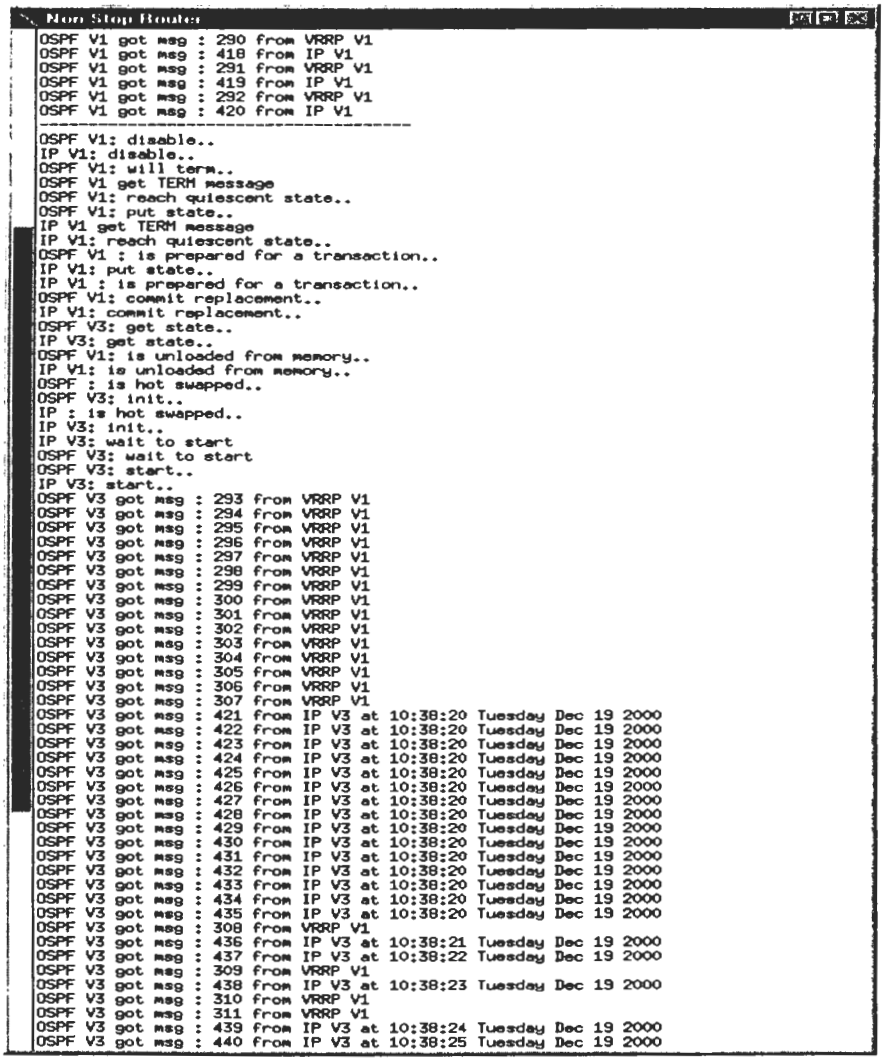


Figure 5.9 Transaction 6

5.3 Discussion of simulation results

The simulation results show that the proposal framework provides the capability for a live software upgrade of an actual non-stop multi-tasking application. Using the framework, the Modules OSPF, IP, VRRP in a non-stop application can be replaced, added and removed dynamically. The simulation also demonstrated the process of concurrent upgrading a group of modules, and proved that an upgrade transaction can be aborted in case of failure and the all or nothing property can be guaranteed.

Moreover, the simulation demonstrated the ability to accommodate multi-version of a message protocol and the backward compatibility features.

Chapter 6

Conclusions

6.1 Summary

The dynamic software upgrade framework provides a practical mechanism for fast, safe and reliable software upgrade at run time.

At the beginning of this thesis, we investigated the related works and addressed the advantages and disadvantages of each strategy, which included a hardware-based approach, a component-based dynamic architecture, the dynamic object–based approach, procedure- level dynamic updating, a process-based approach, and an analytic redundancy based approach. These approaches focus on solving the problems at different levels, but none of them can provide the overall solution to live software upgrade. Hence, the dynamic software upgrade framework was proposed and designed to give a powerful and integrated solution.

Furthermore, implementation was discussed on Chapter 4, which includes loading and creation of modules, concurrency control, queuing analysis, definition of operation primitives, state of machine, and object interface.

Additionally, a simulation demonstrates the powerful functions supported in the framework. Those transactions in the simulation deal with a variety of situations, which may occur in an actual application.

6.2 Contributions

1. This thesis provides an integrated practical framework for live software upgrade. The design of the framework emphasizes the four main areas, involving dynamic architecture and communication model, reconfiguration management, run-time upgrade policy, and the upgrade technique.
2. The Design of a dynamic architecture for on-the-fly module replacement. Indirect addressing, publisher and subscriber communication model, the name service, the version-control repository and the software upgrader form a unique dynamic structure for the framework, which make it possible to add, replace or remove the modules on the fly.
3. Design of an upgrade protocol for module addition, replacement and removal and implementation of an atomic upgrade.
   A two-phase commit protocol is introduced to ensure safe, reliable upgrade. A concurrent upgrade mode is applied for the upgrade transaction to minimize the downtime of application services.
4. Mechanism for maintaining state consistency.
   In order to have the capability of controlling the upgrade transaction and keeping the state consistent, the notion of a module-proxy is used to extend the traditional concept of *port*. A simple and practical approach to keeping the application integrity during the upgrade is analyzed. And the strategy of reaching the stable state is illustrated in the framework as well. Some issues on module dependency and scope change are also addressed.

6.3 Future work

1. To distribute the new version of the software across a network. It is desirable that the new version of the software can be downloaded from a HTTP server. So the software can be either pulled from the web server automatically by an upgrader-enabled application, or the software can be pushed to the application.
2. Security validation is necessary before an upgrade request is processed. Right of access should be defined according to security rules.
3. The current framework focuses on the reliability of the upgrade transaction during application changes. It can be extended to ensure reliability after application changes. The rollback is executed when a new module introduces some faults and breaks some existing functionality. The mechanism can be used for run-time testing. The multi-version of modules may be kept running in the application.
4. The performance of the framework can be measured further. The number of target modules, the number of pending messages in the queue, the size of transmitted state become main parameters that influence the performance. The mechanism of upgrading a hard real-time application by using the framework needs to be investigated.
5. The framework can be extended to a distributed environment as well. In a distributed system, the upgrade protocol, the two-phase commit protocol and the concurrent upgrade mode can be still used for upgrade transaction. The name service and publisher-subscriber communication mode fit the distributed environment well. But some implementation issues need to be addressed. For example, how to order the messages sent in a distributed system, how to transfer the states between two modules running on different machine, etc.
6. Explore the mechanism of garbage collection (dynamic storage allocation for state persistence).

Bibliography

[1] Knight, S., etal. "Virtual Router Redundancy Protocol (VRRP)", RFC 2338. 1998.

[2] Jeff Magee, Jeff Kramer, "Dynamic Structure in Software Architectures," *Fourth SIGSOFT Symposium on the Foundations of Software Engineering,* pp.3-14, San Francisco, Oct. 1996.

[3] Peyman Oreizy, Richard N. Taylor, "On the Role of Software Architectures in Run-time System Reconfiguration," *Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4).* pp61-70, Annapolis, Maryland, May, 1998.

[4] Peyman Oreizy, Nenad Medvidovic Richard N. Taylor, "Architecture-Based Run-time Software Evolution," *Proceedings of the International Conference on Software Engineering 1998 (ICSE '98),* pp177-186, Kyoto, Japan, April 19-25, 1998.

[5] Robert Laddaga and James Veitch, "Dynamic Object Technology," *Communications of the ACM,* Volume 40, Number 5, pp.37-38, Dec.1997.

[6] Daneil D. Corkill, "Countdown to Success: Dynamic Objects, GBB, and RADARSAT-1," *Communications of the ACM,* Volume 40, Number 5, pp49-58, Dec.1997.

[7] Guy L. Steele, "Common Lisp the Language, 2nd edition," *Thinking Machines, Inc. Digital Press* 1990.

[8] Sangal Rajeev. "Programming Paradigms in LISP," *McGraw-Hill,* 1991.

[9] Bruce Eckel, "Thinking in Java," *Prenticall Hall PTR,* 1998.

[10] Mark E. Segal, Ophir Friender "On-the-fly Program Modification: Systems for Dynamic Updating," *IEEE software,* pp.53-65, March, 1993.

[11] Mark E. Segal , Ophir Fieder, "Dynamic Program Updating in a Distributed Computer System," *International Conference on Software Maintenance,* pp198-203, 1988.

[12] Deepak Gupta, Pankaj Jalote, and Gautam Barua, "A Formal Framework for On-line Software Version Change," *IEEE Transactions on Software Engineering,* Vol.22, No.1, pp.120-131, Feb. 1996.

[13] Deepack Gupta, Pankaj Jalote, "Increasing System Availability through On-Line Software Version Change," In *Proceedings of 1993 IEEE 23rd International Symposim On Fault-Tolerant Computing*, pp30-35, Aug.1993.

[14] Lui Sha, "Dependable System Upgrade," *Technical Report, Carnegie Mellon University*, Sep. 1998.

[15] Jonathan E. Cook, Jeffrey A. Dage, "Highly Reliable Upgrading of Components," *Internatinal Conference on Software Engineering 1999*, pp.203-212, *Los*Angeles CA.1999.

[16] Lui Sha, Ragunathan Rajkumar, Michael Gagliardi,"Evolving Dependable Real-Time Systems," *Proceeding of Aerospace Application Conference,* pp.335-346, Vol.1 1996.

[17] Mike Gagliardi, Raj Rajkumar and Lui Sha, "Designing for Evolvability: Building Blocks for Evolvable Real-Time Systems," *In Proceedings of the IEEE Real-time Technology and Applications Symposium,* pp.100-109, June 1996.

[18] Jeff Kramer and Jeff Magee, "The Evolving Philosophers Problem: Dynamic Change Management," IEEE Transactions on Software Engineering, Vol.16, No.11, pp.1293-1306, Nov. 1990.

[19] Umar Syyid, "The Adaptive Communication Environment: ACE, A Tutorial," *Hughes Network Systems*, May, 2000.

[20] Douglas C. Schmidt "Applying Patterns and Frameworks to Develop Object-Oriented Communication Software," *Handbook of Programming Languages*, Volume I, edited by Peter Salus, MacMillan Computer Publishing, 1997.

[21] R. Greg Lavender, Douglas C. Schmidt, "Active Object: an Object Behavioral Pattern for Concurrent Programming," *Proceedings of the 2nd Pattern Languages of Programs Conference*, September 1995

[22] Michael Franz, "Dynamic Linking of Software Components," *IEEE Computer,* Vol.30, No.3, pp.74-81, March. 1997.

[23] W. Wilson Ho and Ronald A Olsson. *"An Approach to Genuine Dynamic Linking,"* Software-Pratice and Experience, *Vol, 21, No. 4, pp.375-390, April, 1991.*

[24] Donn Seeley, "Shared Libaries as Objects," *USENIX Summer Conference Proceedings*, pp.25-37, 1990.

[25] Erich Gamma, Richard Helm, Ralph Johnson, John vlissides, "Design Patterns, Elements of Reusable Object-Oriented Software", Addison-Wesley, 1994.

[26] Douglas C. Schmidt "The Adaptive Communication Environment: An Object-Oriented Network Programming Toolkit for Developing communication Software," *11th and 12th Sun Users Group Conference*, June 1994.

[27] Brent, Welch, "Practical Programming in Tcl and Tk," Third Edition, *Prentice Hall*, Nov. 1999.

[28] John K, Ousterhout, "Tcl and the Tk Toolkit," *Addison Wesley*, 1994.

Appendix

A. The simulation environment

A.1 Design issue

The dynamic software upgrade framework provides a mechanism for fast, safe and efficient upgrade of software applications at run-time. In order to benefit from the framework, all the module implementation class (*VRRPImp, IPImp, OSPFImp* and *TIMERImp*) in the simulation application must be inherited from the base class *ModuleImp*.

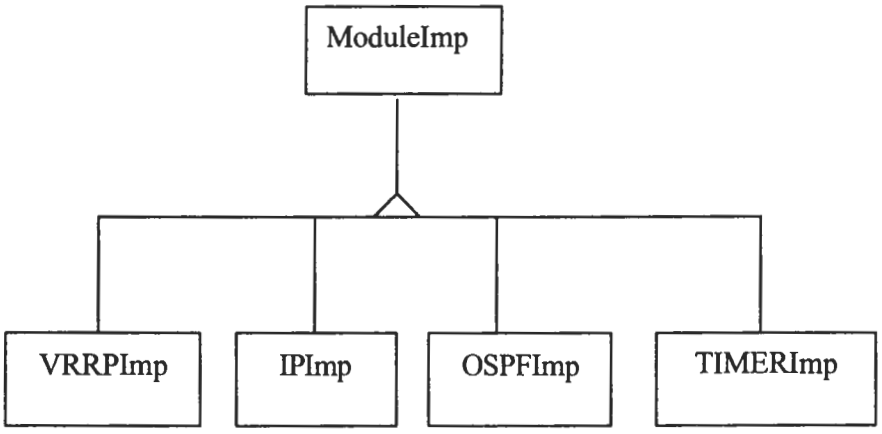


Figure A.1. Class Diagram for module implementation

The following API methods in the base class *ModuleImp* should be overridden by each derived class for reconfiguration management. The method invocation has been encapsulated from the framework (Table A.1)

METHOD	DESCRIPTION
initialize	Implements initialization for the module.

terminate	Terminates the whole task. Automatically called when module finish service ()
service	Dispatches the message request to the appropriate function handler
disable	Disables the module
enable	Activates the module.
put_state	Handles the request for capturing the state of the module
get_state	Handles the request for recovering the state of the module

Table A.1. The common API methods

In addition, each implementation class should implement a create_module() method, which is the factory method for creation of the instance.

A.2 Implementation issue

The simulation application is written in C++, C and developed under Unix TCP/IP environment. Each application specific module implementation must be complied as a shared library, which is placed under the given directory.

A process will run the non-stop router application, while the IP, VRRP and OSPF traffic monitor applications, can be started on the same machine or on adifferent machine on the same network. By telnet and the command line interface, the administrator is able to make the live software upgrade of the router application.

The syntax of the configuration command has been defined in Chapter 3 and some examples have been shown in Chapter 5.

VITA

Surname:Yu Given Names: Lizhou

Place of Birth: Shanghai, China

Education Institution Attended:

University of Victoria	1998 to 2001
Beijing College of Economics	1989 to 1993

Degrees Awarded:

B. Sc.	Beijing College of Economics	1993

Honours and Awards:

Publications:

PARTIAL COPYRIGHT LICENSE

I hereby grant the right to lend my thesis to users of the University of Victoria Library, and to make single copies only for such users or in response to a request from the Library of any other university, or similar institution, on its behalf or for one of its users. I further agree that permission for extensive copying of this thesis for scholarly purposes may be granted by me or a member of the University designated by me. It is understood that copying or publication of this thesis for financial gain shall not be allowed without my written permission.

Title of Thesis:

A Framework for Live Software Upgrade

Author: 

LIZHOU YU

Apr. 13, 2001

(Date)