A Framework for Live Software Upgrade

Lizhou Yu[1] Gholamali C. Shoja Hausi A. Müller Anand Srinivasan[2]
Department of Computer Science
University of Victoria
P.O. BOX 3055, STN CSC
Victoria B.C. Canada V8W 3P6
edward.yu@fnc.fujitsu.com {gshoja, hausi}@cs.uvic.ca anand@sionintl.com

Abstract

The demand for continuous service in mission- and safety-critical software applications is increasing. For these applications, it is unacceptable to shutdown and restart the system during software upgrade. This paper examines issues relating to on-line upgrades for mission- and safety-critical software applications. We believe that a dynamic architecture and communication model provides an excellent foundation for runtime software evolution. To solve the problems mentioned above, we designed and implemented a framework, which addresses four main areas: dynamic architecture and communication model, reconfiguration management, the upgrade protocol, and the upgrade technique. The framework can be used for on-line upgrading of multi-task software applications, which provide multiple mission-critical services.

In the framework discussed in the paper, 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, remove, 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 facilitates the reconfiguration management. A simulation study of the proposed framework was carried out for live software upgrade of several practical applications. The results of the simulation are also presented.

[1] At present with Fujitsu Network Communication, Texas, U.S.A.

[2] Formerly at Nortel networks Ottawa, Canada. At present with Sion International, Ottawa, Canada

1. Introduction

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

The evolutionary change of software is unavoidable due to changes in the environment or in the application requirements that cannot be completely predicted during the design phase, or due to bug-correction or enhancement of functionality. For these mission- and safety-critical applications, it is unacceptable to shutdown and restart the system during software upgrade, since monetary loss, interruption of service, and damage can be caused with a traditional installation process. In the network communications industry, the criteria for high availability require that the services must be provided 24 hours a day, 7 days a week with near 99.99% uptime. 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.

Upgrading a non-stop application is a complex process. The new and the old component may differ in the functionality, interface, and performance. Only selected components of an application are changed while the other parts of the application continue to function. It is important to safeguard the software application's integrity when changes are implemented at runtime. A runtime software upgrade cannot be done at any time, since it may halt or crash the application. The techniques of runtime upgrade are quite dependent on the operating system and the programming language in which the application is written. The ability to deal with failure of upgrade transactions significantly influences the applicability of a live software upgrade technique.

We believe that a dynamic architecture and communication model provides an excellent foundation for runtime software evolution. To solve the problems mentioned above, we designed and implemented a framework, which addresses four main areas: dynamic architecture and communication model, reconfiguration management, the upgrade protocol, and the upgrade technique. The framework can be used for on-line upgrading of multi-task software applications, which provide multiple mission-critical services.

This paper is organized in the following manner: Section 2 compares the related research; Section 3 outlines the design of a dynamic software upgrade framework; Section 4 examines some implementation issues related to the framework; Section 5 describes the simulation results; Section 6 makes the discussion and presents the experience; and, finally, Section 7 summarizes the paper.

2. Related work

This section discusses selected approaches to the problem of live software upgrade.

Hardware-based approach: In a primary-standby system [1], two devices run the equivalent program and back each other up for continuity of service. To perform the update, the first device is stopped at a safe point in the program and simultaneously the second one is started up. After the first one is upgraded, it takes the role of the second device. So the second device is taken off-line and is ready to be upgraded as well.

Component-based dynamic architecture: *Darwin*, proposed by Jeff Kramer and Jeff Magee, is a configuration language for describing dynamic architecture [2]. 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. C2-style architecture is another component-based architecture, which highlights the role of connectors in supporting runtime change [3][4]. Connectors are explicit entities that bind components together and act as mediators among them. Components communicate by passing asynchronous messages through connectors. Connectors provide a mechanism for adding and modifying component bindings in order to support reconfiguration.

Process-based approach: Deepak describes an approach to modeling change at statement level for a simple imperative programming language [5]. The state transfer takes place when the stack is guaranteed to contain no routine, which is to be changed. The replace module then copies the data and stack of the first process onto the second one. The machine registers are copied next.

Analytic redundancy based approach: *Hercules* [6] and *Simplex* [7][8] permit safe on-line upgrading of software despite residual errors in the new components. Analytic redundancy facilitates extensive testing for reliable incremental evolution of safety critical systems. It focuses on how to rollback when a new unit does not satisfy explicit performance and accuracy requirements after replacement. However, it does not illustrate well how to deal with the failure during the upgrade transition.

Distributed object-based approach: In CORBA [9] and COM+ [10], client IDL stubs and server IDL skeletons are generated at the compilation of IDL interface so that a client object can transparently invoke a method on a server object across the network. *Eternal* [11][12] extends the CORBA standard with the object replication and fault tolerance. The method invocation will be handled by a group of objects, so that if in a distributed application one replica object fails or is being upgraded, another object is able to operate normally. The intermediate code can be generated to facilitate the live upgrade after comparison of the versions of class. This approach requires basic CORBA architecture, reliable group communication such as totally ordered protocol, and frequent checkpoint mechanism in order to maintain the state consistency in the object replicas during the running of CORBA applications.

As indicated above, researchers have employed different tactics to solve the problem of live software upgrade. The primary standby method 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. The distributed object-based approach can be implemented via extensions of CORBA standard and object replication. In next section, a framework based on a unique dynamic architecture and an upgrade protocol is proposed, which provides a novel and integrated solution to live software upgrade. This framework is appropriate for multi-task software applications to perform live upgrade at software module level in a centralized environment.

3. A dynamic software upgrade framework

3.1 Overview

Figure 1 depicts a dynamic software upgrade framework, which provides a solution to deal with upgrading non-stop applications at run-time. It can be divided into two parts. The first part is the dynamic configuration service, which includes a Command Line Interface, a Software Upgrader, a Name Service, an Event Manager, a Version-control Repository and a Module-Proxy. The dynamic modules, which consist of module-implementations, constitute the second part. Dynamic modules are those upgradeable components, which can be disabled, enabled, loaded, unloaded, and hot swapped.

3.2 Dynamic Architecture and Communication Model

A dynamic architecture described as follows provides an ability to change module interaction and dependency and notify the rest of software components of the change of a module.

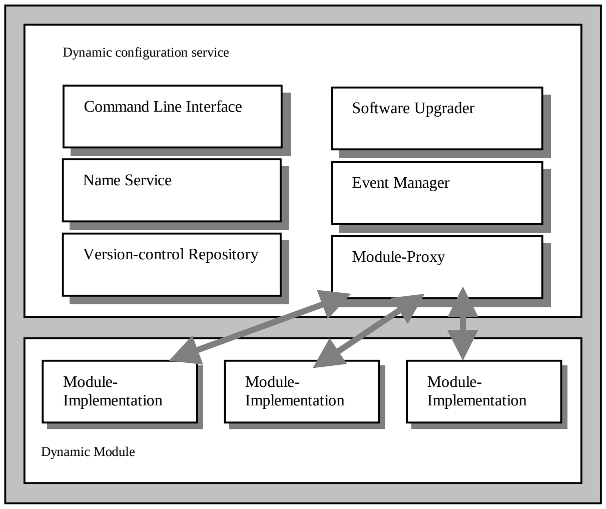


Figure 1. A dynamic software upgrade framework

3.2.1 The addressing problem and the decomposition of modules. Usually, modules communicate with each other through message passing. In direct addressing, the sender needs to know specific destination reference. However, after an existing module is replaced with a 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 instead to a well-known port, which has a shared data structure consisting of queues that can temporarily hold messages. Using indirect addressing can decouple the sender and the receiver, providing great flexibility in dynamically updating the existing modules.

To extend indirect addressing, we split an ordinary module into two parts. One component is a module-proxy restricted to one per module; the other is a module-implementation. A module-proxy is used for minimizing coupling between modules. This idea is similar to the notion of decoupling definition and implementation of a module in some programming languages and distributed architectures. (e.g., Modula-2, ADA, C++, CORBA, COM+). Since the framework operates in a non-distributed environment, the purpose of proxy is to prevent the implementation modules from directly referencing one another during local communication. When a module-implementation is in the Service state, its module-proxy forwards all the incoming requests to it as depicted in Figure 2.

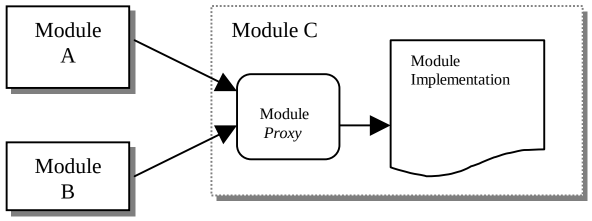


Figure 2. Communication among modules

3.2.2 Module-proxy and module-implementation. A module-proxy is given a module name and is dynamically associated with its module-implementation. It serves as a port forwarding all the incoming requests. In addition, when updating its module-implementation, the module-proxy controls the consistency of state between two versions of the module-implementation, and swaps its reference to the new version. In case of an upgrade failure, the module-proxy enforces recovery and brings back the old version.

A 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:

- It provides a certain amount of functional behavior to participate in a runtime change. To support runtime evolution, a module-implementation must be packaged in a form, such as shared libraries, that can be loaded and unloaded dynamically in a runtime environment.
- There is an input-message queue associated within each module-implementation for asynchronous communication. It internally dispatches the functions based on the received message's type. The interface

dependency among modules lies in the message protocol used.

3.2.3 Name service. Our dynamic software upgrade framework provides a uniform name service where

- any module can be bound to any name (i.e., a string),
- the name service can be used to register and resolve the reference to a module at runtime.

Instead of direct addressing, the reference to a module-proxy is registered with the name service and bound to the name of the module. To send messages to a module, other software modules firstly search the name service for the reference to the module-proxy. And then the messages can be sent to this module via its proxy, which internally appends these messages to the queue of the module-implementation. Finally, its module-implementation further processes the messages. Therefore, removal and replacement of a module-implementation within a module become transparent to other modules.

3.2.4 Publisher and subscriber communication model. The publisher and subscriber model is an event-driven architecture used for communication between modules. A module can be, at the same time, both publisher and subscriber. At any given time, the publishers do not need to know the subscribers and vice-versa. When one module joins or is deleted from an event tag, it does not affect other modules. By holding all the event information and centralizing the management of module communication, the model can reduce the overhead of software evolution at runtime. An event manager contains all the information on registered event tags and interested subscribers. After a publisher tells the event manager to notify its event, the event manager sends an event notification in the form of a message to all the subscribers. With deployment of the model, the upgrade can change the set of modules and modify the interconnection patterns dynamically so that the software can be evolved continually.

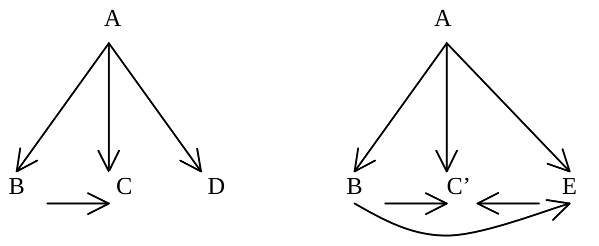


Figure 3. Changes of communication pattern

Figure 3 shows the dramatic changes in the module set and communication pattern before and after software upgrade. The upgrade results in the following impacts: the module C is replaced with a new version, the module D is removed, and a new module E is added. The new module E starts subscribing the events published by the modules A and B. The upgraded module C' subscribes the events not only from the modules A, B but also from the module E.

3.2.5 Version-control repository. In our 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 runtime. The repository provides the software upgrader with an ability to record and hold all the handles of module libraries that can be used to create an instance of the loaded module or clean up the module according to different situations. Moreover, the repository can validate the version and avoid duplicate loading of the same library.

3.2.6 Software upgrader. A software upgrader is normally blocked, waiting for various requests from the command line. Its functionality includes uploading or removing the module library into or from memory, registering the handle of a library to the version-control repository, creating an instance of a dynamic module and registering it with the name service, and finally coordinating with the module-proxy for upgrading the module-implementation. The most important feature is that the software upgrader makes a final decision for committing or aborting an upgrade transaction.

3.3 State Consistency

The state of a module keeps on changing while a software application is involved in different transactions. If a module is replaced, the new module must transform the state of the old module and perform necessary actions to synchronize its internal state with that of the old module. It is important to determine what state information should be transferred from the old one and what time is suitable for checkpointing. This can be roughly divided into a stable and a transient state. To avoid overheads such as a deep copy of the stack and machine registers, transient states should not be considered for checkpointing.

Once a module enters into a kind of stable state called quiescent state [13], the module state can be checkpointed and transferred. Configuration management must give the affected module an opportunity to reach a quiescent state before a change is performed. A replaced module reaches a quiescent state if it does not currently initiate a new request and it does not currently engage in serving a request. To keep the quiescent 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. The simple solution is that all the new incoming request messages are buffered into a message queue while a target module is upgraded. All the buffered messages are passed to the modules being replaced later. As a result, no request message gets lost during updating.

The module-proxy in 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 messages while the software upgrader acquires 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 its module-implementation to reach a stable state.

On the other hand, the module-implementation should terminate its task context once the last termination message from its input queue is received. 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. As the size of a message-buffer queue is critical for the target module to avoid losing pending messages, a provision of the size of queue is provided for module proxies to determine their own need.

3.4 The Runtime Upgrade Protocol

The Runtime Upgrade protocol is an important part in a live software upgrade. To deal with module replacement, creation, and removal, the protocol consists of a *module replacement protocol, a module creation protocol, and a module removal protocol* respectively. These three protocols are described in subsequent sections.

3.4.1 The Module replacement protocol. The Module Replacement protocol describes how to execute an on-line replacement of a module. The protocol can be divided into three phases: the uploading phase, the replacement phase, and the clean up phase.

- **The uploading phase** (Figure 4)

1. The software upgrader receives a replacement command from the application administrator.
2. The software upgrader looks up the command message and validates using the name service if the updated module has been activated and uploaded. Otherwise, it aborts the command.
3. After the validation check, the software upgrader uploads a new module library into the memory according to the command message, and registers the handle of the library with the version-control repository.
4. Then the software upgrader creates an instance of a dynamic module. In addition, it notifies the module-proxy that the current module-implementation will be upgraded.

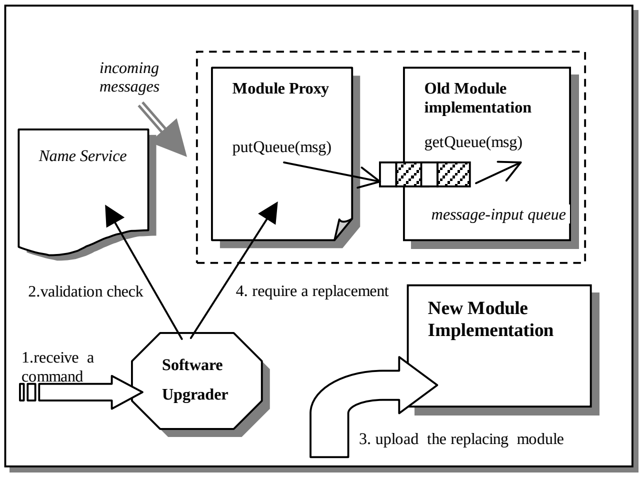


Figure 4. Uploading phase

- **The replacement phase** (Figures 5 and 6)

1. The module-proxy collaborates with its module-implementation to achieve a quiescent state and waits for a notification from its implementation.
2. After reaching its quiescent state, the module-implementation calls its 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. Thus the new module-implementation gets the chance to synchronize its state with that of the old one.
5. The proxy is dynamically associated with the new module-implementation.

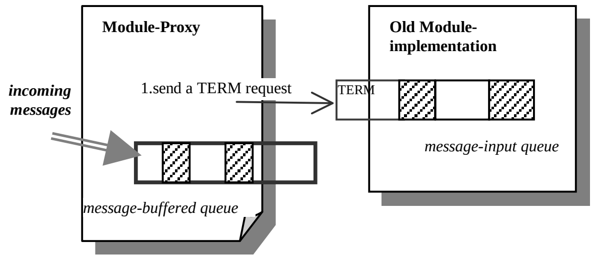


Figure 5. Sending a termination message

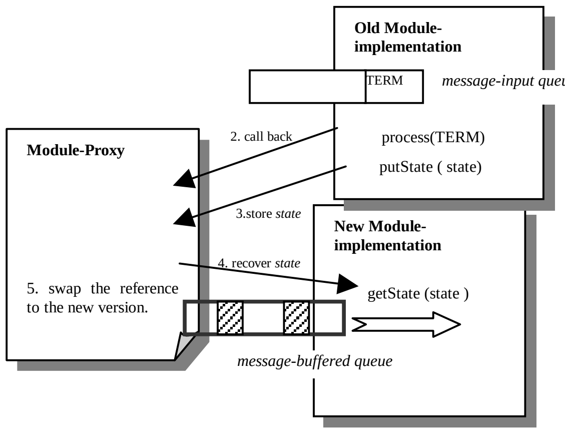


Figure 6. Replacement phase

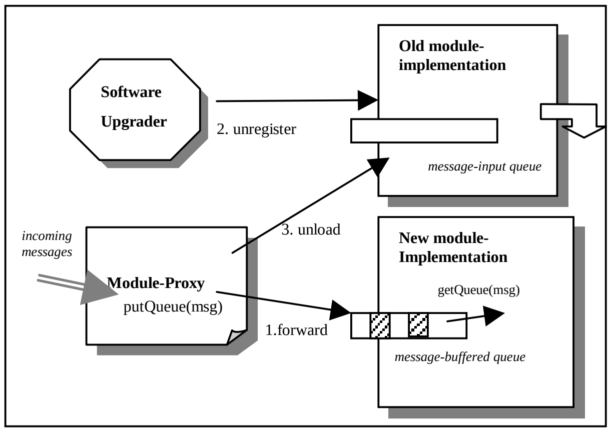


Figure 7. Cleaning phase

In general, not entire state of the module but the state that is necessary for the new version to synchronize with the old version shall be transferred. The protocol requires that the essential state of an application module shall be explicitly specified.

Certain memory blocks or a file is created by an old version of module to stores its state. The module proxy will be responsible for informing the new version of module of the storage address once the old version has completed the progress of storing data. Then a new version of module will retrieve the state from this stable storage.

To speed up the synchronization, a capability for direct transfer of state bypassing the module-proxy, can be implemented since the task of transfer and retrieval of state can be performed simultaneously between two versions. A state manager in the old version of implementation can segment and wrap the state as small data units that will be sent continuously to the new version for synchronization.

A mechanism of state handler implemented by abstract factory pattern can be deployed to solve the problem of state compatibility when transforming state between modules with different representations. A “right” state handler will be created by the new module for transformation of state as long as the version number of the old module is known.

- **The Clean up phase** (Figure 7)

1. The proxy activates the new module-implementation and redirects all subsequent request messages to it.
2. The software upgrader removes the handle of the old module library from the version-control repository.
3. The proxy unloads the old module-implementation from the main memory

3.5 Upgrade Techniques

3.5.1 Atomic transaction protocol. The atomicity of software transitions requires that when a live software upgrade to a group of modules is completed, either all or none of its operations should be carried out. If one part of a transaction is aborted, then the whole transaction must also be aborted. The failure of an upgrade transaction may occur if any target module fails to reach a quiescent state within the bounded time or fails to extract or store its internal state. The validation check is performed within each software module for the versioning, the completion of state transfer and the timeout of upgrade. A *two-phase commit* protocol is designed to achieve the atomic operation in our framework. During a voting phase, every module prepares for the upgrade transaction and returns a vote to the software upgrader based on the validation check. During a completion phase, the software upgrade will decide to commit or abort the transaction according to the votes from every target module.

In case of commitment, the reference to module-implementation will be hot swapped within the module proxy. And the new version will start running at the transferred state. Otherwise, the new version of module-implementation will be removed. And the old version will be recovered at the frozen state.

3.5.2 Concurrent upgrade. When a group of modules is upgraded simultaneously and the two-phase commit protocol is applied, an appropriate strategy of the execution mode should be chosen to reduce service downtime in an application. A concurrent upgrade mode can be used to solve the problem. A master-slave mechanism is introduced to allow concurrent execution. At the beginning, the software upgrader, a master, spawns a slave thread for each target module. In addition, each slave, is associated with a task and is in charge of upgrading a module. Because slaves and their target modules can concurrently commit or abort a command within their task contexts, the performance of an upgrade transaction could be significantly improved.

Figure 8 depicts the sequence of a transaction in a concurrent 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. During the voting phase, two spawned slaves help the *M1* and *M2* modules reach quiescent state and transfer their state. If validation check is successful, each slave will vote YES on behalf of the module. Thus, during the completion phase, the software upgrade will commit the whole transaction including swap of the implementation within the *M1* and *M2* module and addition of new *M3* module. As barriers are introduced to synchronize the action, it allows the upgrader to order the sequence. Finally, all the involved modules are started by the upgrader at the end of the transaction.

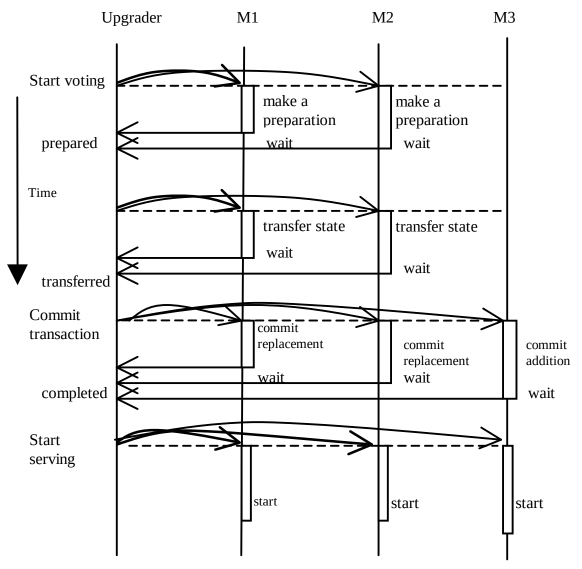


Figure 8. Synchronization in a concurrent upgrade process

3.5.3 A command line interface for reconfiguration management. To fully control operations during application transition at runtime, the application administrator should be given permission to reconfigure a software application via the command line interface. To maximize flexibility of the application transition, a command line interface can be used in two ways: (1) application transition for only one module; and (2) the simultaneous upgrading of a group of modules that depend on each other. A scripting language, such as Tcl/Tk [14], provides powerful capabilities to build such an application-specific command line interface.

4. Implementation environment

We implemented the framework in C++, because a key objective was to develop an extensible and reusable framework. To adapt to different environments and reduce portability problems, *ACE* (Adaptive Communication Environment) [15][16] was used to map the framework onto many platforms, such as several versions of Unix, Linux, and Win32.

Dynamic library facilities provided by the operating systems allow, at run time, loading a module library that has been complied as a shared library. The object-oriented system design further allows creating an instance of a dynamic module at run time. To take advantage of the features described above, dynamic modules satisfy the following implementation requirement.

Each module-implementation is wrapped as a C++ class, which is derived from a base class *ModuleImp*. This Class can be compiled separately as a dynamic shared library that can be loaded into and unloaded from memory at runtime [17][18][19]. The base class *ModuleImp* contains a message queue and some essential functions used for internal manipulation of live software upgrades.

Furthermore, the class *ModuleImp* exports some well-known interface, which can be overridden by its subclasses that implement software modules. Due to inheritance and late binding, the newly created instance of a subclass can be dynamically type cast to its base class *ModuleImp*. Therefore, it is possible to pull out an old version of the module-implementation and plug in a new version at run time.

The module-proxy component is implemented by the class *ModuleType* that interacts with module-implementation internally and exports some public interface for other software modules to communicate with its delegated module. 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 to the module implementation can be hot swapped to a new *ModuleImp* object by the *ModuleType* object.

We found that software design patterns can facilitate the implementation of the framework. Also, mutual exclusion and barrier mechanisms can help us resolve the synchronization problem when a group of modules is upgrade.

5. Simulation results

Our demonstration application based on the framework is called non-stop Router consisting of four modules: *TIMER*, *IP* (Internet Protocol) [20], *OSPF* (Open Shortest Path First) [21], and *VRRP* (Virtual Router Redundancy Protocol) [22], which are concurrently executed and provide multiple services as depicted in Figure 9. On one hand, ICMP (Internet Control Message Protocol) packets, HELLO discovery packets, and HEARTBEAT packets are broadcasted periodically. 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. To simulate a publisher and subscriber communication model, the *TIMER* module acts as a publisher by publishing a broadcast event. Other modules will subscribe to the broadcast event once they are activated. When the *IP* module receives an event message periodically, it broadcasts an ICMP packet to the neighborhood and meanwhile sends a control message to the *OSPF* module. Similarly, the *VRRP* module not only broadcasts a HEARTBEAT packet to nodes but also sends a control message to the *OSPF* module. The *OSPF* module only broadcasts its HELLO packet after receiving an event message. To maintain application consistency before and after a live software upgrade, the content in messages sent by the *VRRP* module and the *IP* module to the *OSPF* is assigned an integer number, called the relay number, in a consecutive format.

Section 5.1 below, illustrates an example of a successful replacement transaction, while section 5.2 shows the scenario of an aborting transaction.

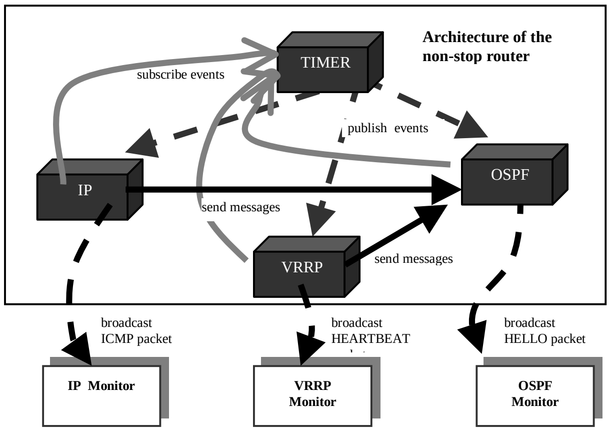


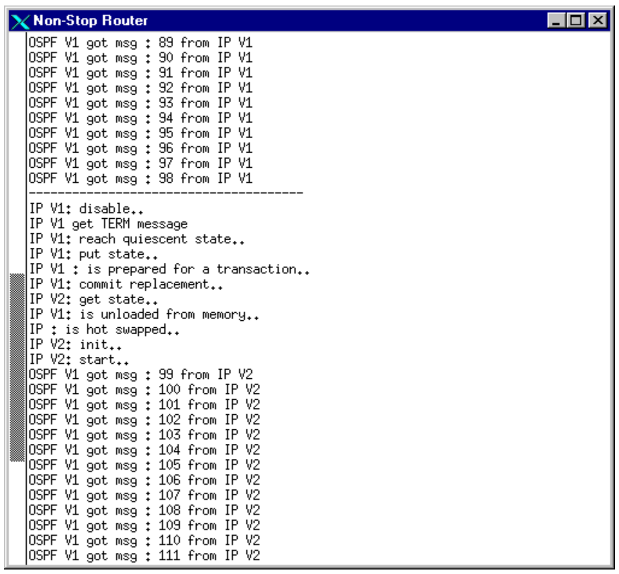
Figure 9. Architecture of the non-stop router

5.1 Transaction One

<u>Configuration command</u>

```
Replace  IP  libIPImp_v2.so
```

Transaction one as depicted in Figure 10 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 context and reaches the stable state. Once Version 1 of the implementation object has checkpointed its state and put its relay number into the state storage, Version 2 of the 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.



```
OSPF V1 got msg : 89 from IP V1
OSPF V1 got msg : 90 from IP V1
OSPF V1 got msg : 91 from IP V1
OSPF V1 got msg : 92 from IP V1
OSPF V1 got msg : 93 from IP V1
OSPF V1 got msg : 94 from IP V1
OSPF V1 got msg : 95 from IP V1
OSPF V1 got msg : 96 from IP V1
OSPF V1 got msg : 97 from IP V1
OSPF V1 got msg : 98 from IP V1
--------------------------------------
IP V1: disable..
IP V1 get TERM message
IP V1: reach quiescent state..
IP V1: put state..
IP V1 : is prepared for a transaction..
IP V1: commit replacement..
IP V2: get state..
IP V1: is unloaded from memory..
IP : is hot swapped..
IP V2: init..
IP V2: start..
OSPF V1 got msg : 99 from IP V2
OSPF V1 got msg : 100 from IP V2
OSPF V1 got msg : 101 from IP V2
OSPF V1 got msg : 102 from IP V2
OSPF V1 got msg : 103 from IP V2
OSPF V1 got msg : 104 from IP V2
OSPF V1 got msg : 105 from IP V2
OSPF V1 got msg : 106 from IP V2
OSPF V1 got msg : 107 from IP V2
OSPF V1 got msg : 108 from IP V2
OSPF V1 got msg : 109 from IP V2
OSPF V1 got msg : 110 from IP V2
OSPF V1 got msg : 111 from IP V2
```

Figure 10. Transaction One

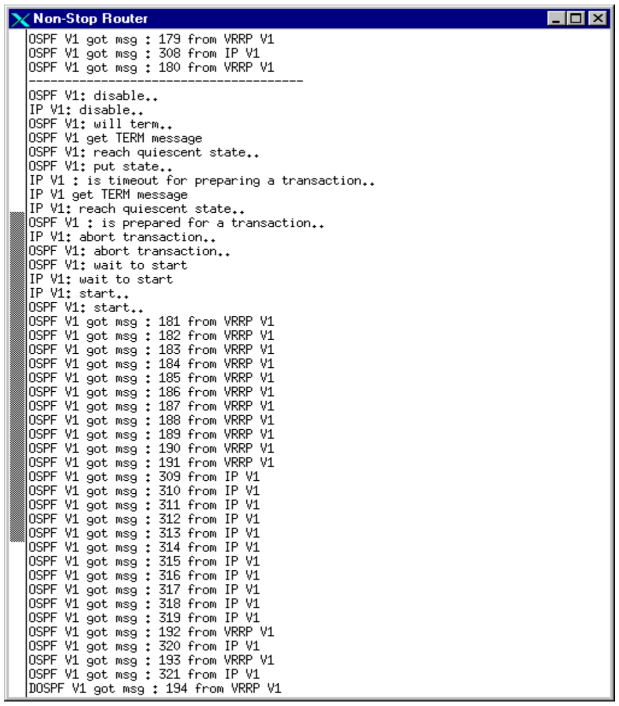
5.2 Transaction Two

<u>Configuration Command</u>

```
SetTimer   2
Syn_Replace    OSPF  libOSPFImp_v2.so
               IP    libIPImp_v2.so
```

Figure 11 depicts the scenario of an aborting transaction. The maximum preparation time is two seconds, which may be shorter than the time taken by one of the modules to prepare for live upgrade. Within two seconds, the *OSPF* module can reach a stable state, return its vote, and attain voting state. However, the *IP* module cannot meet the timing constraint. Thus, the *IP* module returns a "NO" vote to the software upgrader. The software upgrader decides to abort the transaction because one of the target modules votes "NO". Consequently, the newly created implementation objects are removed from memory, and the original implementation objects are automatically brought back into service. Finally, the queue of the buffered message is handed over to the current implementation object. Although the transaction is aborted, no pending messages or new incoming messages are lost.

The simulation results show that the proposed framework provides capabilities for live software upgrade of an actual non-stop multi-tasking application. Using the framework, the Modules *OSPF*, *IP*, *VRRP* in a service-critical 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.



```
OSPF V1 got msg : 179 from VRRP V1
OSPF V1 got msg : 308 from IP V1
OSPF V1 got msg : 180 from VRRP V1
--------------------------------------
OSPF V1: disable..
IP V1: disable..
OSPF V1: will term..
OSPF V1 get TERM message
OSPF V1: reach quiescent state..
OSPF V1: put state..
IP V1 : is timeout for preparing a transaction..
IP V1 get TERM message
IP V1: reach quiescent state..
OSPF V1 : is prepared for a transaction..
IP V1: abort transaction..
OSPF V1: abort transaction..
OSPF V1: wait to start
IP V1: wait to start
IP V1: start..
OSPF V1: start..
OSPF V1 got msg : 181 from VRRP V1
OSPF V1 got msg : 182 from VRRP V1
OSPF V1 got msg : 183 from VRRP V1
OSPF V1 got msg : 184 from VRRP V1
OSPF V1 got msg : 185 from VRRP V1
OSPF V1 got msg : 186 from VRRP V1
OSPF V1 got msg : 187 from VRRP V1
OSPF V1 got msg : 188 from VRRP V1
OSPF V1 got msg : 189 from VRRP V1
OSPF V1 got msg : 190 from VRRP V1
OSPF V1 got msg : 191 from VRRP V1
OSPF V1 got msg : 309 from IP V1
OSPF V1 got msg : 310 from IP V1
OSPF V1 got msg : 311 from IP V1
OSPF V1 got msg : 312 from IP V1
OSPF V1 got msg : 313 from IP V1
OSPF V1 got msg : 314 from IP V1
OSPF V1 got msg : 315 from IP V1
OSPF V1 got msg : 316 from IP V1
OSPF V1 got msg : 317 from IP V1
OSPF V1 got msg : 318 from IP V1
OSPF V1 got msg : 319 from IP V1
OSPF V1 got msg : 192 from VRRP V1
OSPF V1 got msg : 320 from IP V1
OSPF V1 got msg : 193 from VRRP V1
OSPF V1 got msg : 321 from IP V1
DOSPF V1 got msg : 194 from VRRP V1
```

Figure 11. Transaction Two

6. Discussion and experience

We have accumulated some valuable experience with our prototype implementation. We found that a live upgrade performed at a modular level is appropriate for complex software systems. Our framework supports the following features: (1) a run-time evolvable software architecture at a granularity of module, (2) an atomic upgrade transaction without influencing other running modules, (3) maintaining the state consistency during changes, (4) an incremental execution of reconfiguration in soft real time, (5) a concurrent upgrade mechanism.

In some service and/or mission-critical applications, their module behaviors rely on some periodic control messages exchanged between modules. To avoid changing the state machine of software modules due to a live upgrade, an upgrade transaction must be committed or aborted within a timing constrain. Moreover, no control message should be lost during the upgrade.

Scope change is the extent to which the software modules in application are affected by an upgrade. The revision or enhancement of the message protocol causes the changes to the interface of modules. Since a message protocol is defined in communication peer, both modules become replaced modules when their message protocol is changed. The proxy of a module remains same except that the implementation will be upgraded. Such a dependency on a message protocol decides the scope change of target 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. To achieve backward compatibility, two versions of the message protocol should be both accommodated by the replacing module. This will continue until all the modules that use the older version of the message protocol have been updated. Thus, a module is able to create and use the correct message handler to process the messages labeled with a version. In this way, scope change of software module as a result of a live upgrade becomes minimal.

To obtain the benefits of the framework, the software applications should meet the following requirements. 1) The software modules must be written as derived class of *ModuleImp* based on dynamic updating protocol. 2) Messaging and the model of publisher and subscriber shall be adopted for intercommunication. 3) The implementation of software modules shall enable the feature of state transfer. Since the application state is variable, the "right" state handler will be written for a new implantation to transform the state from the old. The framework provides essential infrastructures, binding proxies with module implementations, upgrading the modules, wrapping and transferring the state between the versions of the implementation etc.

As service availability and state consistency are preferred, failure to upgrade the modules that have not reached quiescent condition within some bounded time is an acceptable compromise. The plan of later retry is considered as an alternative.

7. Conclusion

We have presented an integrated practical framework for live software upgrade. The design of the framework emphasizes four main areas: dynamic architecture and communication model, reconfiguration management, runtime upgrade protocol, and upgrade technique. We introduced a unique dynamic architecture which includes indirect addressing, a publisher and subscriber communication model, a name service, a version-control repository and a software upgrader. We designed an upgrade protocol for module addition, replacement, and removal. A two-phase commit protocol was introduced to ensure robust upgrades. A master-slave concurrent upgrade mode is applied to minimize the downtime of services provided by a software application. We implement a mechanism for maintaining state consistency and controlling the upgrade transaction.

We expect that future work in this framework will include performance measurement, adaptation to a distributed environment, and some traditional authentication issues.

8. References

[1] Pankaj Jalote, “Fault Tolerance in Distributed systems” Prentice Hall, 1998.

[2] Jeff Magee and Jeff Kramer, “Dynamic Structure in Software Architectures,” Fourth SIGSOFT Symposium on the Foundations of Software Engineering (FSE), pp. 3-14, San Francisco, October 1996.

[3] Peyman Oreizy and Richard N. Taylor, “On the Role of Software Architectures in Runtime System Reconfiguration,” Proceedings of the International Conference on Configurable Distributed Systems (ICCDS 4), Annapolis, Maryland, May 1998.

[4] Peyman Oreizy, Nenad Medvidovic, and Richard N. Taylor, “Architecture-Based Runtime Software Evolution,” IEEE/ACM International Conference on Software Engineering (ICSE '98), pp. 177-186, Kyoto, Japan, April 19-25, 1998.

[5] Deepak Gupta and Pankaj Jalote, “Increasing System Availability through On-Line Software Version Change,” Proceedings of 1993 IEEE 23rd International Symposium On Fault-Tolerant Computing, pp. 30-35, August1993.

[6] Jonathan E. Cook and Jeffrey A. Dage, “Highly Reliable Upgrading of Components,” IEEE/ACM International Conference on Software Engineering (ICSE ’99), pp. 203-212, Los Angeles, CA.1999.

[7] Lui Sha, “Dependable System Upgrade,” Technical Report, Carnegie Mellon University, Sep. 1998.

[8] 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.

[9] Object Management Group, “The common Object Request Broker: Architecture and specification, 2.2 edition,” OMG Technical Committee Document formal/98-07-01, Feb 1998.

[10] Microsoft Corporation, Various COM documents, MSDN library, 1998.

[11] L. A. Tewksbury, Louise E. Moser, P. M. Melliar-Smith, “Live Upgrades for CORBA Applications using object replication,” IEEE International Conference on Software Maintenance, pp488-497, Florence, Italy, Nov. 2001.

[12] Louise E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. A. Tewksbury, V. Kalogeraki, “Eternal: fault tolerance and live upgrades for distributed object systems,” Proceedings of IEEE information Survivability Conference and Exposition (DISCEX 2000), Vol.2, pp184-196, 2000.

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

[14] Welch, Brent, “Practical Programming in Tcl and Tk,” Third Edition, Prentice Hall, Nov. 1999.

[15] 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.

[16] 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.

[17] Michael Franz, “Dynamic Linking of Software Components,” IEEE Computer, Vol. 30, No. 3, pp. 74-81, March 1997.

[18] 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.

[19] Donn Seeley, “Shared Libraries as Objects,” USENIX Summer Conference Proceedings, pp. 25-37, 1990.

[20] F. Baker, “Requirements for IP Version 4 Routers”. RFC 1812. June 1995.

[21] J. Moy, “OSPF Version 2”. RFC 2328. April 1998.

[22] Knight, S., et al., “Virtual Router Redundancy Protocol”, RFC 2338. April 1998.