Run-time Software Upgrading Framework for Mission Critical Network Applications

Seung-Woo Hong, Seong Moon and Ho-Yong Ryu
ETRI (Electronics and Telecommunication Research Institute), 218 Gajeongno, Daejeon, Korea

Keywords: Live Software Upgrade, Run-time Upgrade, Mission-critical.

Abstract: In mission critical and safety software applications such as internet infrastructure, telecommunication, military and medical applications, service continuity is very important. Since for these applications it is unacceptable to shut-down and restart the system during software upgrade, run-time software upgrade techniques, which are deployed for online maintenance and upgrades without shutdown the system, can meet the demand for high levels of system availability and service continuity. However, upgrading an application while it is running without shut-down 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. Various researchers have employed different tactics to solve the problem of run-time software upgrade such as compiler-based methods, hardware-based method, and analytic redundancy based. In order to ensure a reliable run-time upgrade, we designed and implemented a software framework based run-time software upgrading method, which has the ability to make runtime modification is considered at the software architecture-level. In this paper, we present the software component architecture for run-time upgrade and software upgrade procedure, and then show the implementation results.

1 INTRODUCTION

In information technology, high availability refers to a system or component that is continuously operational for a desirably long length of time. The high availability for continuous service is important in safety critical software applications such as internet infrastructure, aero-space, tele-communication, military and medical applications, since monetary loss, interruption of service and unpredictable damage can be caused with any moment of software failure. However, software change is unavoidable, because the software requirements change, a bug is bound or optimisation and enhancement of functionality is discovered. To upgrade the software for these reasons, halting execution of the existing software and restarting with new one is inevitably involved, and those upgrading approach results in software outage and service interruption (Jeff, 1996; Tewksbury, 2001).

The purpose of run-time software upgrade technique is to dynamically upgrade the behaviour of a running software system without the software outage and service interruption. Various researchers have employed different tactics to solve the problem of run-time software upgrade such as component based (Jeff, 1996; Peyman, 1993), process based (Deepak, 1993), analytic redundancy based (Jonathan, 1999; Mike, 1996), distributed object based (Tewksbury, 2001; Louise, 2000), dynamic module based (Michael, 1997; Wilson, 1991; Donn 1990; Drossopoulou, 2002; Yu, 2002), and compiler patch based (Chen, 2007; Fahmi, 2008; Neamtiu, 2006; Makris, 2009; Chen, 2016). But, in the real fields most general and widely used way to achieve the run-time software upgrade is through the use of replication that is based on redundant hardware (Deepak, 1993). The basic idea is that two machines are available, 'A' and 'B'. The both machines run same application software, and software of machine 'A' runs as active role to provide actual application service while 'B' runs as standby role. The state of 'A' and 'B' should be synchronized by some kind of synchronization mechanism such as check-point service. If the software needs to be upgraded, standby B is brought up running the new software

Hong, S-W., Moon, S. and Ryu, H-Y.
Run-time Software Upgrading Framework for Mission Critical Network Applications.
DOI: 10.5220/0006355007270732
In *Proceedings of the 19th International Conference on Enterprise Information Systems (ICEIS 2017) - Volume 2*, pages 727-732
ISBN: 978-989-758-248-6

while active ‘A’ still provides the application service. After upgrade of standby ‘B’, switch-over of the role take place and B runs as active role to provide new application service with new software. The problem of redundant hardware basis is the expensive solution and, it still has retaining problem of transferring the state information, more precisely, how to extract relevant state information from old software, and transformed to be compatible with and injected into new software (Yu, 2003).

In this article, we propose the run-time software upgrading method that avoids temporary interruption caused by a software upgrade by allowing the system to be updated on-the-fly without hardware redundancy. We focus on software architecture based approach, and designed and implemented run-time software upgrade framework. The proposed framework provides dynamic software component architecture, communication model between dynamic software components and run-time module upgrading procedure.

2 RUN-TIME UPGRADE

To support run-time software upgrade, we designed run-time software upgrade framework. In the framework, a software component is defined as single software process that performs role of a specific application role. Inter-process communication between the software components is done via message passing based on socket or message queue. This section describes the internal software architecture of dynamic component and gives details of software upgrade procedure.

2.1 Software Component Architecture

Figure 1 shows internal architecture of dynamic software components that supporting run-time software upgrading. The dynamic software component consists of two modules. The first one is main task module, which is persistent and unchangeable part of a component while the component process is running. The other is a dynamic implementation module that performs application specific functions and can be updated dynamically. Detailed characteristics of the modules are described as follows:

Main Task Module: The module consists of singe task thread, and the thread serves as main loop to process incoming events such as request message from other component, timer event caused by time expiration and signal event from kernel. All events are buffered by event-queue, and the single thread dispatches the events one by one and processes it sequentially. As mentioned before, this module is unchangeable part of component, and it controls upgrading procedure of user dynamic implementation module, that is, it swaps old user module with new one when updating dynamic implementation module. In addition, it manages the user module data to preserve the consistency of state after update.

Dynamic Implementation Module: This module, as dynamic module, includes all the application specific implementation. It can be updated on the fly while main task module is still running. The module consists of task call-back function and user module implementation. The task call-back function includes a set of statically defined call-back functions such as process-message, process-timer and process-signal. Basically, it connects task thread and module implementation. The task thread re-maps reference of the task call-back functions whenever dynamic module is changed, and then the connection between main task module and dynamic implementation module can be retained. To support run-time evolution, we make the module in form of packaged shared library in order to load and unload dynamically in a run-time environment.

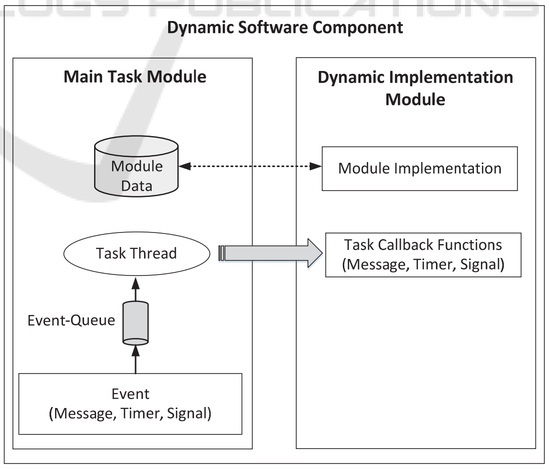

Dynamic Software Component
Main Task Module
Dynamic Implementation Module
Module Data
Module Implementation
Task Thread
Task Callback Functions (Message, Timer, Signal)
Event-Queue
Event (Message, Timer, Signal)

Figure 1: Architecture for dynamic software component.

Figure 2 depicts an example of message communication between dynamic component A and B, and shows how the component processes incoming request messages as bellow.

1. Component 'A' sends a request message via certain kind of inter process communication method. In case of the example, we use socket with specific port number.
2. A socket bound to the port number is triggered in the kernel, which notify to component.
3. The task thread inputs the socket event into FIFO event-queue.
4. The task thread also dispatches one event from event-queue, and if the event is type of message, it calls process-message call-back function in task call-back functions module.
5. The called process-message call-back function calls a user defined message processing function, which is defined by module implementation.

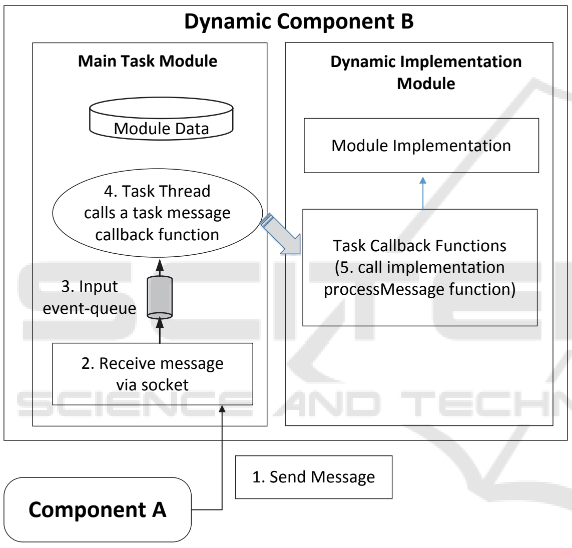

Dynamic Component B
Main Task Module
Module Data
4. Task Thread calls a task message callback function
3. Input event-queue
2. Receive message via socket
Dynamic Implementation Module
Module Implementation
Task Callback Functions (5. call implementation processMessage function)
1. Send Message
Component A

Figure 2: Example of message processing.

Processing procedures for timer and signal event almost same as the message except that user can register and un-register certain amount of time to be expired. To maintain data consistency between old and new user module, we preserve user module data inside main task module and the task thread provides reference of data to user implementation module when the user module is loaded. Therefore we don't have any synchronization of module data between old module and newly upgraded module in our upgrade framework.

2.2 Software Upgrade Procedure

To deal with module replacement, creation and removal, we define a sequence of operations between main the task module and the dynamic implementation module as blows:

- INIT: the task tread calls this operation to user module when dynamic user module is initially loaded in order to make the new user module to initialize application specific functions.
- TERM: the task tread calls this operation to user module when upgrade is needed or when process need to be terminated. The dynamic user module should stop its work immediately.
- HOLD: the task tread calls this operation to user module before starting upgrade, which requests that the user module should be in quiescent state.
- RESTART: the task tread calls this operation to user module after loading the new user module, which inform that the new user module restart its function with restored data reference.

Figure 3 shows the procedure for unloading old dynamic implementation module and loading new dynamic implementation module. First, to upgrade to new dynamic module, new dynamic module need to be compiled as shared library and located in software repository. And then, user can trigger upgrade procedure by command line interface which sends upgrade request message to target software component. The task thread of the target component starts upgrade procedure as blows:

1. The task tread of the target component checks the incoming message, and if the message is upgrade request, then it starts dynamic upgrade procedure.
2. The task thread calls TERM operation to stop the old module, and subsequently it calls HOLD operation to make the old module to be in quiescent state.
3. The task thread unloads old module from memory. Function of target software component is temporarily stopped while the upgrading, but no incoming events are lost since those events will be queued in event-queue.
4. After then, the task thread load new module package into memory. Remapping procedure between main task and new user module is described in Figure 4.
5. After loading the new module, the task thread starts remapping procedure because address reference of task call-back functions of new module might be different from old

one. The task tread relinks all the defined static call-back functions of new user module.

6. The task manager restores reference of user data by sending the reference through remapped task call-back functions.
7. The task tread calls RESTART operation to allow new module to start its application functions. And the task thread starts to dispatch queued event from event-queue.

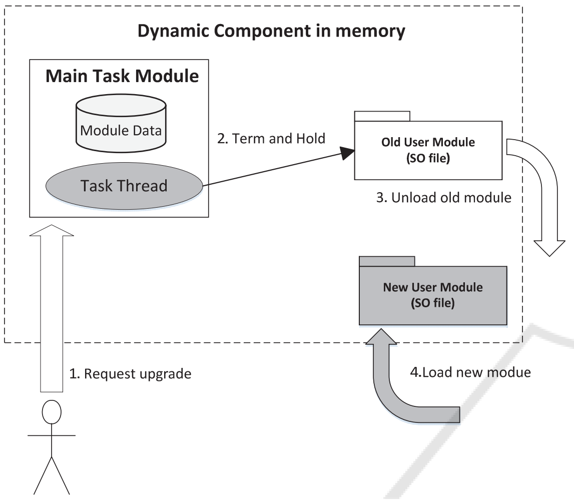


Figure 3: Unload old module and load new module.

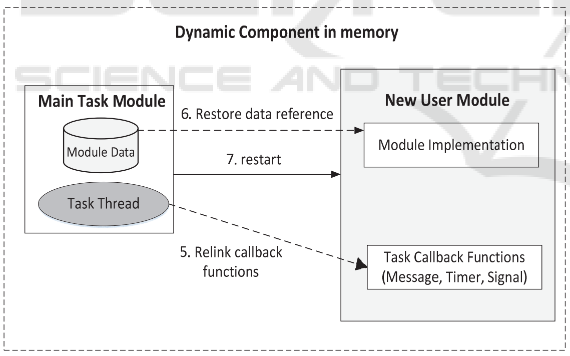

Dynamic Component in memory
Main Task Module
Module Data
Task Thread
6. Restore data reference
7. restart
5. Relink callback functions
New User Module
Module Implementation
Task Callback Functions (Message, Timer, Signal)

Figure 4: Re-mapping between main task and user module.

3 IMPLEMENTATION RESULTS

3.1 Implementation Environment

As mentioned above, we implemented the main task module as main process of program that is main part of our run-time software upgrade framework. To make main event loop that is used for processing every event such as IPC message by network sockets, timer expired event, signal triggered by kernel, and so on, we utilized the well-known libevent software library. The libevent provides a mechanism to execute a call-back function when a specific event occurs on a file descriptor including socket or after a timeout has been reached, and it also support call-back triggered by signals and regular timeouts. As shown in Figure 5, the main task module forms single threaded event loop by libevent and every event from outside of dynamic component are dispatched through event-dispatch function which subsequently calls the event-receive functions such as receive-message, receive-timer and receive-signal function in dynamic module. In our implementation framework, the main task module also has various libraries API including IPC message, memory handling, logging, and so on, therefore user who wants make dynamic component only take care of application logic itself in dynamic module.

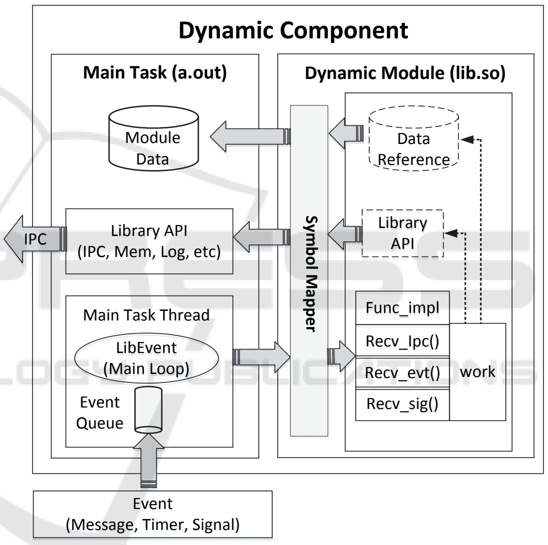


Figure 5: Implemented architecture of dynamic component.

We realize the dynamic implementation module as type of shared library because the shared library are intended to be shared by executable modules and loaded into memory at load time or run time rather than being copied by a linker when it creates a single monolithic executable program. The only way to get into dynamic module is through implemented event-receive function in dynamic module, then every request or event for dynamic module is coming though event loop in main task module. As mentioned earlier, the main task module constructs symbol mapping table called symbol mapper between main task module and dynamic module whenever it loads dynamic module. We have three symbol mapping table in symbol mapper, the first one is event-receive symbol mapping that is used for connection while event-dispatch, second one is library API symbol mapping and third one is permanent module data reference mapping. For dynamic linking between main task module and dynamic implementation module, POSIX's DL (dynamic loaded) libraries such as dlopen, dlsym and dlclose are used. The dlopen loads new data and code of dynamic implementation module into a component process's address space, and the dlsym provides a mechanism to locate functions in the dynamic implementation module by name, and dlclose unload old dynamic implementation module. As a result, main task module and dynamic module can be separately compiled and produce each binary module. The main task module forms an executable program and the dynamic module takes the form of a shared library as shown in Figure 5.

3.2 Test Results

Test application is network router system that is one of core element of internet network infrastructure. A router is connected to two or more date lines from different networks. When a data packet comes in on one of the network lines, the router reads the address information in the packet to determine the ultimate destination. To decide the destination of packet, all of router maintains its own routing table that is created by routing protocol software such as RIP protocol. In case of upgrading the routing protocol, network operator should shut down the old routing protocol and restart the new routing protocol, which results in losing current routing table and network connection is down until the new routing protocol newly constructs new routing table. The objective of the test is that routing protocol does not lose its routing table after upgrading with our run-time software upgrading framework.

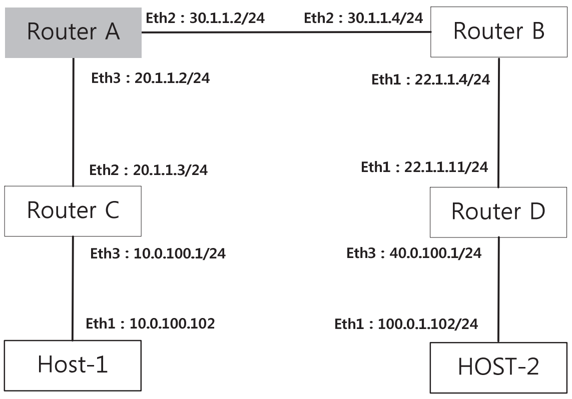
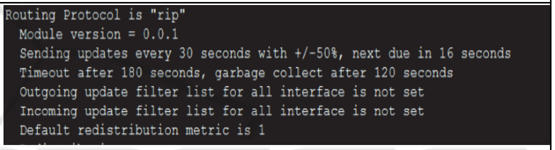


Figure 6: Test network environment.

Figure 6 shows test network environment. There are RIP router A, B, C and D, and two hosts Host-1 and Host-2. Four RIP routers provide a connection path between two hosts. All of routers already make their own routing table and connection check is done by testing ping between Host-1 and Host-2. Furthermore we continue to send ping messages to Host-2 from Host-1 to confirm a connection path during upgrade. The ping messages will be stopped if network connection is down, which means RIP protocol in route A loses its routing table during upgrade. We have the CLI (Command Line Interface) that is used to configure RIP protocol's parameters or show current status of the protocol. Table 1 shows CLI commands and the results of status of the protocol. The current module version of RIP is 0.0.1 and new version is 0.0.2 as shown in Table 1. We

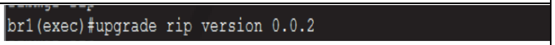
Table 1: Test results.

CLI shows old RIP module (librip_0.0.1.so).



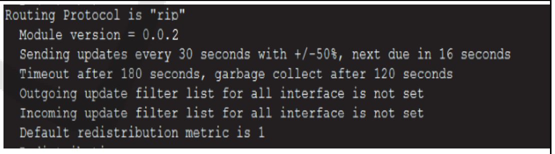
```
Routing Protocol is "rip"
  Module version = 0.0.1
  Sending updates every 30 seconds with +/-50%, next due in 16 seconds
  Timeout after 180 seconds, garbage collect after 120 seconds
  Outgoing update filter list for all interface is not set
  Incoming update filter list for all interface is not set
  Default redistribution metric is 1
```

CLI to upgrade to version 0.0.2.



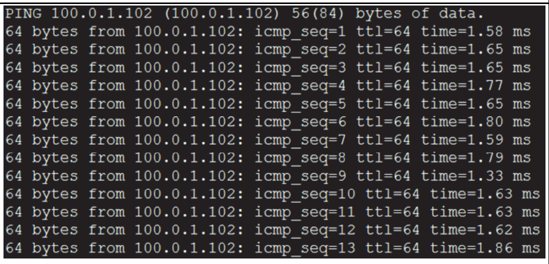
```
br1(exec)#upgrade rip version 0.0.2
```

CLI shows new RIP module (librip_0.0.2.so).



```
Routing Protocol is "rip"
  Module version = 0.0.2
  Sending updates every 30 seconds with +/-50%, next due in 16 seconds
  Timeout after 180 seconds, garbage collect after 120 seconds
  Outgoing update filter list for all interface is not set
  Incoming update filter list for all interface is not set
  Default redistribution metric is 1
```

ping result to Host-2 during upgrade.



```
PING 100.0.1.102 (100.0.1.102) 56(84) bytes of data.
64 bytes from 100.0.1.102: icmp_seq=1 ttl=64 time=1.58 ms
64 bytes from 100.0.1.102: icmp_seq=2 ttl=64 time=1.65 ms
64 bytes from 100.0.1.102: icmp_seq=3 ttl=64 time=1.65 ms
64 bytes from 100.0.1.102: icmp_seq=4 ttl=64 time=1.77 ms
64 bytes from 100.0.1.102: icmp_seq=5 ttl=64 time=1.65 ms
64 bytes from 100.0.1.102: icmp_seq=6 ttl=64 time=1.80 ms
64 bytes from 100.0.1.102: icmp_seq=7 ttl=64 time=1.59 ms
64 bytes from 100.0.1.102: icmp_seq=8 ttl=64 time=1.79 ms
64 bytes from 100.0.1.102: icmp_seq=9 ttl=64 time=1.33 ms
64 bytes from 100.0.1.102: icmp_seq=10 ttl=64 time=1.63 ms
64 bytes from 100.0.1.102: icmp_seq=11 ttl=64 time=1.63 ms
64 bytes from 100.0.1.102: icmp_seq=12 ttl=64 time=1.62 ms
64 bytes from 100.0.1.102: icmp_seq=13 ttl=64 time=1.86 ms
```

dynamically upgrade RIP component from old librip_0.0.1.so to new librip_0.0.2.so through CLI command in RIP router 'A'. We can see that Host-1 still receives ping response from Host-2, and connection path is preserved, which means that RIP router ‘A’ still has its routing state consistency after changing the dynamic module.

4 CONCLUSIONS

Service continuity is very important in mission critical and safety software application such as internet infrastructure, telecommunication, military and medical applications, since these applications above, it is unacceptable to shut-down and restart the system during software upgrade. The purpose of run-time software upgrade technique is to dynamically upgrade the behaviour of a running software system without the software outage and service interruption. In this article, we present a run-time software upgrading method based on software architecture. We proposed the software framework for dynamic software module architecture and run-time module upgrading procedure. Also, we implemented the proposed scheme and show results of run-time upgrading via network router software. In future work, we will focus on further enhanced features such performance measurement and dealing with the case that user data structure is changed in run-time.

ACKNOWLEDGMENTS

This research was supported by the IT R&D program of MSIP (Ministry of Science, ICT and Future Planning) / IITP (Institute for Information & Communication Technology Promotion). [R0101-16-0070, Development of The High Availability Network Operating System for Supporting Non-Stop Active Routing]. This research was supported by the National Research Foundation of Korea (NRF-2012R1A1A2044460).

REFERENCES

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.

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.

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, August 1993.

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

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.

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.

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.

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

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.

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

S. Drossopoulou and S. Eisenbach, Manifestations of Dynamic Linking, The First Workshop on Unanticipated Software Evolution (USE 2002), Málaga, Spain, June 2002.

L. Yu, G.C. Shoja, H.A. Muller, A. Srinivasan. "A Framework for Live Software Upgrade", Software Reliability Engineering, 2002, ISSRE 2003, Proceedings. pp. 149 - 158.

H. Chen, J. Yu, R. Chen, B Zang. "POLUS: A POwerful Live Updating System", 29th International Conference on Software Engineering, May 2007, ICSE'07.

S. Fahmi, H. Choi, “Life Cycles for Component Based Software Development”, IEEE 8th International Conference on Computer and Information Technology Workshops, July 2008, CIT Workshop 2008.

I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, “Practical dynamic software updating for C,” in Proc. ACM SIGPLAN Conf. Program. Language Design Implementation, 2006, pp. 72–83.

K. Makris, R. Bazzi, “Immediate multi-threaded dynamic software updates using stack reconstruction”, Proc. Conf. USENIX Annu. Tech. Conf., 2009, p. 31.

Kristis Makris, Rida A. Bazzi, Immediate multi-threaded dynamic software updates using stack reconstruction, Proceedings of the 2009 conference on USENIX Annual technical conference, p.31-31, June 14-19, 2009, San Diego, California.

G. Chen, H. Jin, D. Zou, Z. Liang, B. B. Zhou, H. Wang. "A Framework for Practical Dynamic Software Updating", IEEE Transactions on Parallel and Distributed Systems, April 1 2016, pp. 941-950.