Asian Journal of Information Technology 5(7): 767-771, 2006
© Medwell Online, 2006

Hot Swapping in Component-Based Software Systems State of the Art

Saleh Alhazbi and [1]Aman Jantan
Department of Computer Science, Qatar University, Doha-Qatar
[1]School of Computer Science, Universiti Sains Malaysia

Abstract: This study presents a review of the most recent approaches for replacing a component in component-based software systems at runtime. It investigates common problems with updating software dynamically : system consistency, state transfer, type of updating and safety

Key words: Survey, component-based systems, dynamic updating

INTRODUCTION

As computer systems are applied to more and more aspects of our life, the complexity of software systems is increasing significantly. Traditional methodology of software development does not cope with such complexity. On the other hand, Component-Based Development (CBD) is being increasingly adopted as a mainstream approach to software systems development. In CBD, the software is built by integrating prefabricated components rather than developing everything from scratch. Different definitions of software components were formed; Szyperski[1] defines a software component from a structural perspective as a unit of composition with contractually specified interfaces and explicitly context dependencies only. A software component can be deployed independently and is subject to composition by third parties. Brown[2] defines a component as an independently deliverable piece of functionality providing access to the services through interfaces.

As its software nature, component-based software systems like conventional systems need to be updated over time for different reasons such as fixing bugs, upgrading its components, or adapting the system in response to its environment's changes. Traditionally, software modifications require shutting down the system, update the system and restarting it. This approach is not suitable for systems that require 24/7 availability such as banking or telecommunications systems, or critical systems such as air-traffic controllers. As a result of that, such systems require dynamic updating, which means swapping between different versions of system components without having to terminate the execution of the whole application. Basically, hot swapping with component-based systems means disconnecting a component at run-time and connection a new version of that component. Hot swapping is a subset of dynamic runtime software evolution, which includes runtime reconfiguration, dynamically adding/deleting/replacing a component. Hot swapping is limited to replacing a component with a new version and it has the same meaning with on-line software replacement.

MOTIVATIONS

In CBD, the emphasis is on modular architecture so the component-based systems seem to have highly modularity. As a result of that , they are relatively well suited for dynamic updating by replacing a component with a new version. This section presents some of motivations for building hot swappable systems:

- Updating high availability systems is a fundamental challenge because stopping the system and installing the new version then restarting it- is not acceptable as it might have extreme consequence. Telephony systems, financial transaction systems and air traffic Control systems are few examples of software systems that require high availability.
- Banks can lose as much as US$2.6 million per hour and brokerages as much as US $4.5 million pre hour from down time[3]. In a survey about the cost of downtime in the year 2001[4], 54% of all participating companies stated that each hour of downtime would cost the company more than $50K and 8% said that each hour would cost over $1M. Of all the companies, 4% estimated that the survival of the company would be at risk if the downtime lasted less than one hr and 39% suspected that downtime lasting up to one day would put the survival of the company at risk.
- In the area of fault tolerance, updating component-based system dynamically let the system continues work despite the presence of a fault. When a fault is

Corresponding Author: Saleh Alhazbi, Department of Computer Science, Qatar University, Doha-Qatar

detected; dynamic updating is used to mask a component failure by swapping to error-free one, so the whole system would not fail.

- For best performance, hot swapping feature can be used to build autonomic, self-diagnosing systems which can improve their performance by using different algorithms in different situations and switching from one algorithm to another according to environmental conditions. Significant variations in resource availability should trigger architectural reconfigurations, component replacements[5].

Background: Updating software system while it is running is not a new area of research, it can be tracked back according to the unit of replacement (procedure, class, component). The first research in this area was conducted by Fabry in 1976 who described a system in which implementations of abstract data types can be replaced on the fly[6].

With moving to producers-based development, Segal and Frieder[7] developed Procedure-Oriented Dynamic Update System (POUDS) that allows incremental update of procedures in a running program. In POUDS, many versions of a procedure can coexist in separate segments and inter-procedures are used to map calls from one segment to another. Gupta[8] established a theoretical base for program dynamic updating. According to that, he built a formal framework for modeling changes to running programs and developed a system for online version changes based on the notation of process and process states. His model was developed for on-line changes to programs written in sequential procedural language. In his approach, the function in a program was the unit of updating; when a new version to be installed, a new process is created with the new software. When the time is suitable, the system transfers the state of the old process to the new process and kills the old one. For the suitable time of state transfer, Gupta defines a formal model for validating transferring states between an old and a new version. Hicks[9] presented a dynamic updating approach for a C-like language with much focus on type-safe updating using patches that contain both updated code and code for state transfer between old and new version. Those patches are generated automatically. Lately withobject-oriented programs, hot swapping means replacing some of the classes that composed the application dynamically while it is running. We can classify approaches for building dynamic classes into two types:

- Proxy pattern approaches. Those approaches require no support from runtime system. The dynamic classes are wrapped by a proxy using language features in order to be able to redirect any referencing when there class updating. Examples on this kind: Hjálmtýsson's approach for building dynamic classes in C++[10], Also approach proposed in[11] which used java interface features to build dynamic classes.
- Runtime-dependent approaches. Such approaches require special support from the runtime. For example [12,13] describe techniques to update java applications by modifying Java Virtual Machine (JVM) in order to fully support dynamic class.

Problems with software hot swapping

System consistency: Updating an application dynamically means replacing some of its components at run-time, but the problem is what to do if that component is involved in some operations when updating request is fired. For that reason, replacing a component in the system can not be done arbitrarily. This requires that component should be deactivated before start updating process. Other components that depend on that one also need to be deactivated in order not to request for service while it is out of order. Therefore, architecture of the system should be presented in some way in order to track dependency between components. Another solution for that is to queue all requests to that components. When updating process is finished, the new version handles those requests.

State transfer: Any approach for replacing a component of the system dynamically should preserve the sate of the system, so when a new version of a component takes place at the system, it should start from the state that old version stopped at. This requires a suitable representation of internal state of a component and a transfer function to map state of old version to the new one. It is clear that finding the relation between the old version and new one can not be fully automated. For example if the old class represents the triangle by three points and the new one represents it as two lines and an angle, one can never expect a software tool to find this relationship fully automatically[14].

Type of updating: When updating a component, the type of updating can be categorized into two categories based on the difference between old version and new one: a) Implementation updating: the new version of the component has exactly the same interfaces as the old one, but one or more of implementation of its services have changed. Usually such updating is for performance reasons. b) Interface updating: in this case, the updating is not only with service implementation. Rather, the new component has different interfaces compared to the old ones. This includes adding, deleting interface or modifying the structure of previous interface. With this type of updating, compatibility between client component (component that requests a service) and the provider one might be broken. One solution is to build a third component acting as an adapter which implements the interface required by the client and performs a translation to the interface the provider implements[15] .

Performance: Currently, there is no programming language that fully supports features related to dynamic updating. It is a problem of type-safety of programming languages. It is clear that the more flexible the type system, the more power is available to carry out actual changes. On the other hand, these powers come at a cost of reduced security[16]. As a result of that, developing hot swappable systems with current programming languages is achieved by using a wrapper and reflective approach. This indirection causes an overhead and drops off system performance.

Safety and fault tolerance: Originally, the goal of hot swapping of software components is to keep the system running continuously. Therefore, it is not acceptable at all that the new version of a component crashes the whole system. The problem with updating running system is that no test phase to verify the changes to the system. In order to benefit of any approach for dynamic updating, it should support a way of fault tolerance or rollback recovery to restore the original system.

Recent approaches: Although there are different approaches to build dynamic updatable software system, only review here most recent approaches based on component-oriented paradigm.

DCUP: In this approach[17,18], an application is a tree-like hierarchy of nested components. A component is divided into a permanent part and a replaceable part. The permanent part contains a Component Manager(CM) and wrappers of the component. The replaceable part contains a Component Builder (CB), functional objects and subcomponents of the component. Orthogonally, with respect to the nature of the operation provided, the component is divided into its functional part and control part. Updating a component means replacing its replaceable part by a new version of this part at run time. When there is a dynamic update process, the CM using wrappers to lock their targets and sends an update request to the CB; the CB stops the execution of all functional objects, save their states to the disk and destroys the replaceable part; the CM downloads and instantiates the new version of the CB, the new CB builds the component (functional objects and sub-components) and retrieves state. The disadvantage of this approach is using wrappers to create indirect link between objects which decrease the application performance. Furthermore, because the application is a tree-like model of components, so if there is an update for a component in the top level, it requires all the application to be redeployed.

Dbeanbox: DbeanBox[19] defines a framework for dynamic updating in component-based system by associating an updateable component to one or more other components that may replace it at runtime. This relation between updateable component and its replaceable ones is expressed in an XML format which also includes :

- Constraints that should be satisfied to replace new component with old version.
- Script that describe the mapping between old and new components, as well as the mapping between their methods.
- Constraints or actions to be performed after updating the component.

This framework allows for example to replace a component by another one, either having the same interface or not (interface mapping facilities are supported), it allows also to change the system architecture by changing the connections between components, by adding or removing components, it guarantees some consistency of the reconfigured system by assuring the state transfer between the old and the new component, passivating and activating components as necessary. When the adaptor is passivated, the received events are stored in a waiting list. The stored events are stamped, A stamp is a number (the current time) that specifies the order of the stored event. Passivate a Bean can be performed by passivating all its source adaptors. When the Bean is activated, the stamp allows to fire the stored events in the correct order[20].The shortcoming of this approach is that the replacing components and their relation to the replaced one should be defined in advance which limits the dynamic updating feature to those predefined components.

S-Module approach: Gang Ao[21] and Ning Feng[22]have proposed a proxy-pattern-based approach for dynamic updating applications written in Java. In their approach, a program is composed of swappable and non-swappable modules. Swappable modules are those they can be

Table 1: Comparison of hot swapping approaches in component-based systems

``	Consistency	State transfer	Type of updating	Safety	Performance
DCUP	Before updating, all methods executions are ended and during updating, the wrapper blocks any access to the component.	During updating process, the component saves its state to the disk and when the new version takes action, it initializes its state from the disk	Implementation	No	Indirect access through wrappers decrease the performance
DbeanBox	Components are connected via adapters that passivate and activate the components. Events are queued during updating.	XML-based description of state transfer function.	Both implementation and interface.	No	Using adapters to control communication among components causes performance overhead
S-Module	The hot swapping can only occur when the old S-Module is in idle state.	This approach does not explicitly provide a explicitly provide a states between versions.	Both implementation and interface	No	Using proxy and reflection suffers a performance penalty.
SEESCOA	When there is an update request, controller component sends freezing message to the specific component to ensure consistency of the system	A new version has necessary code to interpret and transform the state of the older version	Implementation	No	SEESCOA is special run-time system and there is no study regarding its performance

replaced dynamically. A swappable module consists of an S-Proxy and an S-Module. The S-Proxy represents a wrapper around an S-Module so any method invocation to the S-Module must go through the corresponding S-Proxy. Java reflection is used to invoke the methods provided at the new S-Module. This indirection link facilitates the ability of S-Module to be replaced at runtime. The swappable application has a swap manager take care of the hot swapping transaction. The hot swapping can only occur when the old S-Module is in idle state. The new S-Module could have a different interface to the old S-Module.

The weakness of S-Module approach is using Java reflection to invoke the new methods at the new S-Module which makes the application suffers a performance penalty. Also, the programmer has to code the S-Proxy. Moreover , this approach can only support incremental functional modification and extension. The technique used to handle interface change cannot support decremental functional modification. This means that a new S-Module has to keep all the interfaces its corresponding old S-Module has provided.

SEESCOA: The SEESCOA project[23], Software Engineering for Embedded Systems using a Component-Oriented Approach, has developed a modeling methodology for building embedded applications, supported by a composition tool and a run-time component system. Components exchange messages in SEESCOA via connectors and every messages is sent asynchronously. The interaction between two components is made through protocols that specified syntactic level, semantic level, synchronization level and QoS level. Such interaction dose not support updating the interface of a component. In SEESCOA, there is a controller component, always present in the environment. When there is an update request, Controller component sends freezing message to the specific component to ensure consistency of the system, instantiates a new version of the component, then transfers the state from old version to the new one. After transferring the state, the new version is relinked to the system and the old version can be safely removed from the system. In SEESCOA, the new version is responsible for interpreting and importing state of old version.

CONCLUSION

In this overview of the state-of-art in hot swapping of component-oriented systems, we discussed the most recent approaches. Those approaches are reviewed in relevant to four important aspects of any dynamic updating techniques : consistency, state transfer, type of updating and safety. Although the main goal of hot swapping is to let systems run continuously, None of the existing approaches give much attention to system safety after swapping of a component is achieved. Table 1 describes those approaches according to those four criteria.

REFERENCES

1. Szyperski, C., 1999. Component software: Beyond object-oriented programming, Addison-Wesley.
2. Alan. W. Brwan, 1997. Background information on CBD", SIGPC.
3. Group, Y., 2002. How much is an hour of downtime worth to you? Must-Know Business Continuity Strategies, pp: 178-187.
4. Eagle Rock Alliance, Ltd., 2001 Cost of downtime online survey results, Fetched.
5. Fabio Kon, 2000. Automatic Configuration of component-based distributed systems. PhD Thesis - Department of Computer Science, University of Illinois at Urbana-Champaign.
6. Fabry, R., 1976. How to Desing A System in Which Modules an be Changed on the Fly, in proceedings of International Conference on Software Engin., IEEE-CS Press, pp: 470-476.
7. Mark, E.S. and O. Frieder, 1993. On-the-fly program modification: Systems for dynamic updating. IEEE Software, pp: 53-65.
8. Gupta, D., 1994. On-line software version change. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur.
9. Michael Hicks, 2001. Dynamic software updating. PhD thesis, Department of Computer and Information Science, University of Pennsylvania.
10. Hjálmtýsson, G. and R. Gray, 1998. Dynamic C++ classes-A Lightweight mechanism to update code in a running program, In Proceedings of the USENIX Annual Technical Conference, pp: 65-76.
11. Orso, A., A. Rao and M. Harrold, 2002. A technique for dynamic updating of Java software. Proceedings of the IEEE International Conference on Software Maintenance (ICSM 2002), Montreal, Canada, pp: 649-658.
12. Malabarba, S., R. Pandey, J. Gragg, E. Barr and J. Barnes, 2000. Runtime support for type-safe dyamic java classes. In the Proceedings of the European Conference on Object-Oriented Programming.
13. Ritzau, T. and J. Andersson, 2000. Dynamic deployment of Java applications. Presented at the Java for Embedded Systems Workshop, London, United Kingdom.
14. Vandewoude, Y. and Y. Berbers, 2005. Component state mapping for runtime evolution, In Proceedings of International Conference on Programming Languages and Compilers, pp: 230-236.
15. Eskelin, P., 1999. Component interaction patterns, On line Proc,6th Annual Conference on the pattern languages of programs.
16. Ebraert, P. and Y. Vandewoude, 2005. Influence of type systems on dynamic software evolution, In the electronic proceedings of the International Conference on Software Maintenance (ICSM'05) Badapest Hungary.
17. Plasil, F., D. Balek and R. Janecek, 1997. DCUP: Dynamic component updating in Java/CORBA environment. Ech. Report No. 97/10, Dep. Of SW Engineering, Charles University, Prague.
18. Plasil, F., D. Balek and R. Janecek, 1998. SOFA/DCUP: Architecture for Component Trading and Dynamic Updating, in the proceedings of ICCDS'98, Annapolis, Maryland, USA, IEEE CS Press.
19. Ketfi, N.B. and P.Y. Cunin, 2002. Dynamic updating of component-based applications, SERP'02, Las Vegas, Nevada, USA.
20. Ketfi, A. and N. Belkhatir, 2004. Open framework for the dynamic reconfiguration of component-based software. The International Conference on Software Engineering Research and Practice, Track on Team-based Software Engineering (TBSE 04), Monte Carlo Resort, Las Vegas, Nevada, USA.
21. Ao, G., 2000. Software hot-swapping techniques for upgrading mission critical applications On the fly, masters thesis, System and Computer Engineering Department, Carleton University.
22. Feng, N., 1999. S-Module Design for Software Hot Swapping, Master Thesis, System and Computer Engineering Department, Carleton University.
23. Vandewoude, Y. and Y. Berbers, 2004. Supporting runtime evolution in SEESCOA. J. Integrated Design and Process Sci., Transactions of the SDPS, 8: 77-89.