Dynamic upgrade of software

Mika Karlstedt
University of Helsinki

Abstract—**Part of any self-healing system must be a mechanism that allows the software in the system to be upgraded. It is simply not realistic to assume that there would be no bugs in software nor any need to add new functionality. This paper studies some work that has been done in solving the problems related to the dynamic upgrade (or update) of the software. We assume that the system has to serve the clients while the upgrade is in progress. Some of the work is based on the assumption that small amount of downtime is acceptable, while others try to get the upgrade finished with no downtime at all. None of the solutions are quite perfect though, so there is still quite some work to be done.**

I. Introduction

There are always reasons to update software. Some times bugs need to be fixed. Some other time new functionality needs to be included. Some times the software requires restructuring to make maintaining the software easier or improving the performance. Dynamic update (or upgrade) in this paper means a method which allows us to update a running system to a newer version of the software while the system is at the same time serving clients. It is acceptable that the users see small decrease in the responsiveness or in the quality of service, but it should never be bad enough to annoy them.

There is at least three ways to achieve this. We can stop the current server and start a new version of the server. Or we can use hardware and software redundancy to have more than one unit serving the clients. We can then bring them down for the upgrade one at the time, while other units provide the required service to the clients. Or we can dynamically update the software itself on-the-fly.

The drawback of the first method is that there is a short period when the system is not responding. If that is acceptable, we have the advantage that this method is the easiest to implement and has the least amount of side-effects aside from the small amount of downtime. The drawback of the second method is that it requires extra resources to provide the redundancy, but it provides the only solution with absolutely no downtime if performed correctly. On top of the extra resources, it is also possibly much more difficult to implement. If the servers are stateless, then there is only small increase in complexity. The drawback of the third method is that there is still a small time frame when the server is not really providing service. In some cases the system is able to maintain the connections and sessions with the clients. On top of that in the long run there is a decrease in the service because the updated version suffers a small performance penalty from the updates. And lastly it also increases the complexity of the software, although parts of that can be hidden from the developer. On the positive side it requires no extra resources and still allows the continuous service even when the service is session based.

This paper will concentrate on the problems that are encountered with the component based approaches, but will also show that the dynamic update can be used without components by showing a method to create upgradeable programs using C-like language. The chapter II introduces some of the problems encountered with the update process. Some chosen solutions to these problems are introduced in chapter III. The paper ends with the conclusions and some remarks of what I think are the areas where future work should be done.

II. Difficulties of the updating process

Component based development is an area where the dynamic update should work nicely. The basic idea behind the components is, that they are developed by different users in different locations to provide different functionalities or services. The users can create new services by combining different components. As the components are independent entities, it should be possible to just substitute a component with a newer updated version of itself and continue working as normal. Unfortunately the reality is not quite so pretty.

A major problem with any update process is that the system that is updated cannot be used while it is updated. While the application code itself is deterministic, the code does not react well if we change the code in the middle of the execution. Similarly we cannot change the data structures while they are used, as the results would be unpredictable. It is possible to update code that is not in execution during the update. Most methods that allow the system to be operational, while being updated, require that there are update points where the component (or the function or the data structure) is not in use. We can update the component in those points. Whenever the component is next used, it will be the new updated version. The problem is how to find those update points.

Version control is another problem that has not been solved adequately. Let us an example to show which kinds of problems we may have. Figure 1 shows a very simple use case. I have created a component *My Component* version 1.2 that uses two other components *Component A* version 1.4 and *Component B* version 2.2. They in turn use the services of a fourth component *Component C* version 0.8. Then a next version of the *Component A* is released, and for some reason I want to use it. Let us say that it fixes a bug that has been annoying me. Unfortunately the bug fix comes with a price; parts of the interface has changed. Can I just install the new version and continue or do I need to fix my component? If there is no changes to the function calls/method invocations that *My Component* uses then I can indeed just install the new component and carry on, but there is no way to know that

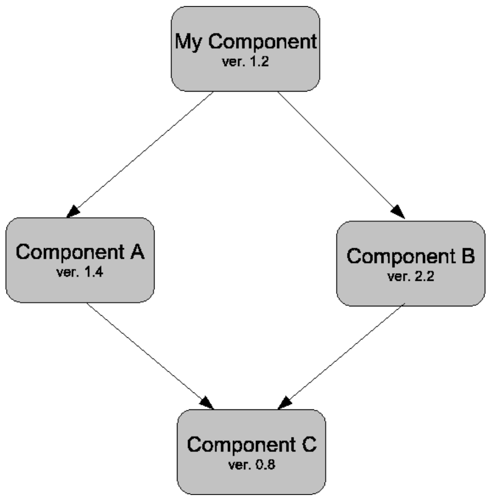

Fig. 1: Simple example of component usage

simply by checking the version numbers. Another problem arises if the new *Component A* requires a newer version of the *Component C* and *Component B* on the other hand does not work with the newer version of the *Component C*. There should be a way to install both of these components at the same time without them interfering with each other?

Most systems have adopted some kind of versioning scheme that tries to solve parts of the first problem. The schemes work by using multipart version numbers like A.B.C.D. When first or second part (A or B) is changed, the API or the interface is also changed. This implies that components implemented against the older versions do not work anymore. Simple bug fixes, that do not alter the interface, change only the third or fourth part (C or D) of the version number. This works, but in some cases it causes the implementer of the client components to fix their components that do not need fixing. That is manual work that would be better left to the software, except that no software exists that could do it.

Redundancy introduces some problems too. Some times there exists many different versions of the same component, and we need to keep track of which is the newest version. That happens especially in cases where components are developed by different people in different places. Another problem is that when we start upgrading the redundant copies of the software, we need to keep track of which copies have been upgraded and which have not. And in some cases copies running old copies cannot communicate with copies running newer copies, which makes the situation even trickier. Once again it would be nice to leave the dirty details to a software. There are indeed attempts to solve the problem and I will introduce one in chapter III-D.

III. Some solutions to chosen problems

The chapter II presented some problems and now is time to present solutions to the aforementioned problems. None of these solutions solve all the problems and indeed there is still lot to do before upgrading software is as easy and reliable as it should be.

A. *Keeping track of different versions*

Stuckenholz's [1] paper is a survey of the State of the Art in version control. He goes through the methods that different systems use to handle the problem using different versions of the same component. The basic idea behind component based approaches is that users can replace existing components with new implementations of the same component. If the user cannot change the component to a newer version, the user is basically using old non-component based approach, where new versions of the libraries require rewriting the application or at least parts of it. We could expect that some of the component based systems would have invented some way to deal with the problem, it is after all at the very core of the component based systems.

Unfortunately there is only two systems in wide use that have solved even a small part of the problem, one is the dynamic shared libraries in *nix like operating systems using ELF-binaries and the other is .NET. Unfortunately the only problem that these systems solve is the problem of keeping more than one version present at the same time without affecting existing applications that use older version.

Shared libraries solve the problem in the linking time. The ELF-binaries are not complete as such. There is the application code but for the shared libraries there are only the symbolic names of the library calls that the binary requires. When the binary is executed, the task is delegated to the linker[1]. The linker loads the software into the memory and then it resolves the required or used symbols from the binary and finds the corresponding entries from the correct libraries. The linker automatically uses the newest binary compatible version of the library present and substitutes the symbolic references with the correct addresses. The linker has to map the required part of the library into the address space of the loaded binary. The system works quite well and is the oldest working solution. So far there is no system that could provide any better solution.

Microsoft uses similar method in their .NET environment but with some differences as the .NET environment is object oriented and maps more nicely to the component paradigm[2]. In the .NET case the object are collected into packages which are called assemblies. The solution works by adding meta data to the .NET assemblies which is used to decide which other assemblies need to be loaded. That alone is not enough as the assembly can only tell against which assemblies it has been developed. Therefore the new assemblies can replace the class loader of the old assembly with a custom loader that loads the newest version instead of the version specified in the assembly.

Java itself has also a limited support for the versioning by allowing developers to add meta data to the classes themselves. The meta data can then be used to decided which class to load. However the decisions need to be done by the developers.

[1]This is not the same linker that is used in the compilation process

[2]Note that Windows XP cannot handle more than one version of any shared library (called DLL).

The language only provides the means to deliver the data but puts no meaning to the data itself and can therefore make no decisions based on it.

B. *Predicting upgrade compatibility*

McCamant and Ernst [6] have developed ways of predicting whether it is safe to upgrade a component. It is easy to ensure that the components provide the same interface, but much more difficult to ensure whether the components have the same behavior or in other words whether the two components are functionally equivalent. Their solution works by creating an operational abstraction of the components and comparing the old and the new version. The solution tries to predict whether the new version would behave correctly. The operational abstraction was created using an open source tool Daikon[9]. The Daikon analyzes the code to find out all the preconditions that must be fulfilled before the component is invoked. It also analysis all the post conditions that hold after the component is invoked. Once the operational abstractions has been developed, the next step is figuring out whether the two are compatible. They use reasoning to figure whether the new component has the same or stronger operational abstraction than the current version. If it has, then it is acceptable to install the upgrade, if not, then it is flagged as unsafe operation.

The advantages of this method, is that is uses automated tools for analysis and is thus less error prone than manual inspection. It is also possible to use the method even without source code, by using a testing framework for finding out what the operational abstraction is. The testing framework is actually built into Daikon, as it runs the software it analyzes inside a debugger. It can then check the values of any variable it deems important before the execution of any specific function, during the execution and after the execution. Of course this requires that the Daikon has a suitable test input that it can feed to the application in study. A side effect of that is that Daikon takes into account the context where the component is used. It will only take into account those parts of the component that are actually used by the application. In other words, if the application uses only a subset of the methods/functions of the component, then the system checks that the used methods/functions behave in the same way in both versions.

The drawbacks are that the system cannot always reach a conclusion, in which case the user still needs some other methods to validate the usability of the upgrade. Also when the upgrade is actually fixing a bug, it is effectively changing the behavior of the component thus changing the operational abstraction, and the method flags even those as dangerous upgrades. Fortunately the system specifies where the problem is, so it is usually possible for the person responsible for the upgrading process to find out that the upgrade is safe.

C. *Automatic loading of the correct classes in Java*

Barr and Eisenbach [2] have created a method for ensuring that the application always uses the newest version of the Java class even when the development is distributed and there are many different developers in different locations. The idea is that the custom class loader created the by the project retrieves always the newest version of the class so that the developers do not need to worry about whether they are using the newest version or older versions.

Java uses byte code to ensure that the code can be run in any computer. The machine independent byte code helps with the distributed systems because all the computers need not be similar. The Java Language Specification introduces a concept of binary compatibility. Two classes that are binary compatible can be used interchangeably. Basically two classes are binary compatible if they have the same methods with the same interface. It is possible to introduce new methods and new fields and maintain backward compatibility. There are quite few different requirements and restrictions that are introduced in the Java Language Specification and the interested reader should check the specification.

The work of Barr and Eisenbach allows one to replace one class instance with another binary compatible version. Unfortunately there are limits to the modifications we can do, if we want to maintain binary compatibility. It is of course better than nothing but hardly a satisfying solution. It seems that Java requires new features if a more advanced replacement algorithm is to be designed.

D. *Automating the update in redundant cluster*

Solarski and Meling [5] tackled the problem of automating the upgrade process in clusters. The purpose of having redundant hardware is to ensure that there is no downtime, but it also means that there is many nodes that need to be upgraded. While it is possible to upgrade them manually it soon becomes both burdensome and error prone. If the user upgrades too many replicas at the same time, it is possible that the clients find the cluster unavailable. It is also possible to forget to upgrade some node if the upgrade is not automatic. In cases where the nodes use services of other nodes some further problems are possible.

Their solution is to create an automated upgrade method. They made some simplified assumptions. The upgrade process of a single node is atomic, and while the node is upgrading, it does not serve the clients. Also the newer version can serve the clients that require older versions of the server. And there must also be a way to transform the state of the old version to the new version. Also other possible upgrades do not interfere with the upgrade in progress. And lastly the interval between upgrades is so long, that at every time there is at most two separate versions *v* and *v+1* of the software running in the cluster. Two versions exist only while the upgrade process is running, once it has finished, there is just one version. This means that it is not possible to upgrade a node from version *v* to version *v+2*.

The upgrade process starts by multicasting the upgrade file to every node. The process uses reliable multicast to ensure that every node receives a copy of the update. The process is truly distributed, meaning that there is no centralized node controlling the update, instead every node uses the same algorithm. The algorithm assumes that the nodes can be ordered in a canonical order by some criteria. The algorithm divides the nodes into two groups, one group that is running the old version *v* and the other group made of nodes already running the next version *v+1*. The node checks whether it is the first node in the group of not yet upgraded nodes. If it is, the node knows that it can upgrade itself, which is does by stopping itself and starting the new version. It then joins back to the group and the new state of the server is transferred to it.

Part of the upgrade implementation is GCS (Group Communication System) that is responsible for reliable multicast. It is also used to transfer the states of the old nodes to the newly joined nodes. The creator of the upgrade has to create a transformation function that can be used to transform the old state to the new state system used in the upgraded version.

The algorithm eases the work of upgrading the cluster while requiring only efficient cluster wide IPC and a method to represent the state of the node. The cluster wide IPC is not a problem as any effective cluster needs it in any case, so the only problem is making sure that there is a way to represent the state of nodes. The state is naturally application specific, and if the application is stateless, then there is no need to have a presentation for the state.

E. Dynamic update in C-like languages

It is possible to create application in C-like language that can be updated while the application is running. Hicks and colleagues have created two different approaches to solve this problem. The first [7] was called pop corn and the second [8] was called Ginseng. I will introduce the latter with little more details as it shows how to create upgradable software. And in any case they did use the lessons learned in the first system when designing the next system.

Their design for the dynamic software updating (DSU) is based on principle that the system should satisfy three criteria, which are:

- DSU should not require extensive modifications to normal software implementation. In other words, it should be possible to concentrate on creating good software and let the underlying framework ensure that the software will be upgradable.
- DSU should allow all kinds of updates. In other words it should be possible to change not only the body of the functions, but also the signature of the functions i.e. the parameters and return values. Finally it must also be possible to change the data structures.
- The last requirement is that the updates themselves should be easy to write and it should be easy to ensure that they are correct.

First the developers create the application with no eye towards the updating. When the application is finished, the source code is compiled with a special compiler[3]. The end result is both the executable and prototype and versioning data. The executable is delivered to a special runtime system, which is in charge of both deploying and upgrading the software.

When a new version is finished, it is delivered to a patch generator, which will receive the prototype and versioning data produced in the first compilation along with the source code. The result is a patch file that is compiled to create a dynamic patch, which can be delivered to the runtime system. The runtime system installs the patch to the running system without stopping the system.

The compiler uses function indirection to make upgradeable software. For every function the compiler generates a global variable which is a function pointer to the correct function. In the first version the pointer is pointed to the first version of the function. When a new version is introduced the pointer is redirected to the new version. Similarly all data structures are wrapped inside an indirection. On top of that a transformer function is defined that can be used to transform an old version of the data type to a new type. It is used during the update process to update the existing data structures to new versions so that we can remove all of the old code. In most cases the creation of the transformation functions is completely automated. Only in some cases do the implementer need to write some tricky parts manually.

Loops require some additional thought. The task is very simple if we move all of the loop code inside a function. Then we can simply change the function pointer to a new version of the loop. One of the parameters to the loop function will be a structure that has pointer to every variable that the loop function uses. Because the loop function has different scope than the loop, it is impossible to access any variable that has local scope in the loop from the loop function.

It is not enough to just create the patch or the application, it is also important to find out the points where the updating is safe to do. Here we have the same problem as in other cases before, it is not safe to modify the code when it is running. Fortunately, it is often possible to find safe upgrading points. For example many server applications use an infinite loop where the server waits for something and then reacts to it. Then the loop ends and the server starts the next iteration of the loop. The end of the loop is a natural point for upgrading. The programmer needs to tell about those upgrading points by making a call to a specific DSU_update-function, which will do the upgrading if needed.

There are some difficult situations which require manual handling. One of them is the void-pointers. Often the function expects a void-pointer as a parameter which it then casts into the correct type. If the compiler cannot solve some cases properly, it creates an error message and leaves the problem to the programmer.

The method is not quite fool proof as it cannot solve all the problems but requires some manual help from the programmer. But in general most of the code can be generated automatically. The upgrade process itself is very fast and the initial overhead for the application is minimal. Unfortunately the overhead escalates bigger when more upgrades are installed.

[3]During the implementation and debug phases the software is of course compiled with a standard compiler, the special compiler is used once the application is ready to be deployed.

The papers did not elaborate on which would be the most likely reasons for it. My personal idea is that the most likely reason is poorer memory management, as the unused code is never released, and it can affect the overall performance quite substantially in the long run. There is nothing that would prevent the system from releasing the unused memory, but in modern operating systems the situation is not quite so simple. The memory management is based on paging[4] and therefore the process can only acquire and release a whole page at a time. If even one byte of the page is used by some old function that is still used, then the whole page has to be kept in memory. A solution could be to implement cleaning patches, that collect all the used functions into a single block of code. That would make it possible to release the old blocks. In real life everything is not quite so simple, but that should at least improve the performance quite drastically in the long run.

F. Updating the operating system

Operating systems have bugs like any other software and therefore need to be updated. But unlike the applications it is not possible to load a new version of the software into the memory and then just transfer the control from one version to the next version. The operating system is by its nature an event driven software. It waits for events to happen and then reacts to them. This itself is not a problem, because many server process works similarly, but there is always many concurrent execution threads [5] active. And as there may not be any bookkeeping data available we cannot know how many of these threads are active. In general it is not possible to reach a quiescent state inside the kernel without actually stopping the kernel. And this is true even to any component the kernel might be made of.

Let us study what would happen if we were to upgrade an operating system kernel. Let us start by upgrading the code. So we replace the existing code with the new version and transfer the control to the new kernel. What do we do with a kernel thread of the old version that has issued a disk transfer and is waiting for it to finish? Clearly it is active though blocked. We cannot terminate it, because we do not know what changes it has made to the data structures. If it is not allowed to finish the execution, we have a real risk of corrupting the data structures of the kernel. We cannot just replace the body of that function with the new function as then the system would execute the first half of the old function and the second half of the new function. We could solve the problem by letting the old code exist as long as there are threads running the old version, but all new instances would use the new version. But what do we do if the new version changed some kernel data structure like. Now we have two threads that must share the data structure but they both use different kinds of structure! There is no easy way out.

I came up with two different ways of doing this. Chen [3] and the team came up with a novel idea of using virtualization and Baumann [4] and colleagues developed an experimental operating system K42 using object oriented methods. They created a way to do a hot swap on the objects which in practice allows the dynamic update of the operating system.

The use of virtualization solves many of the upgrading problems quite elegantly, as the operating system is not really in control of the system anymore. Instead the virtualization system controls the software and it is therefore possible to temporarily stop an instance of the operating system, while it is upgraded. The upgrade selects a single function to be upgraded at a time, and upgrades it. Because the virtualization layer controls also the memory, it is able to control the use of the kernel data structure. If the data structures are modified in the upgrade process, the virtualization layer puts write protection on to the memory used by any data structures. Then it transform the data from the old structure and copies it to the new structure. The result is two different copies of the same data, one used by the old versions and the other used by the new versions. Whenever there is a change in the data, the virtualization reacts to it, because the write triggers a write protection exception. It can then do the correct adjustment to the other data structure. The end result is that both old and new versions can run concurrently and share the same data though their representation is a bit different. In time the old versions will be finished and can be removed from the system.

K42 on the other hand treats operating systems as made of components or more accurately objects. To help the task of upgrading the OS, they used the factory pattern for creating new instances of the objects. The factory pattern is one of the well known programming patterns used in the object oriented programming. When it is used, the user does not create new instances of the objects. Instead a factory is created, which has a method for creating new instances of the class. It helps in many ways, first there is only one instance of the factory so it is much easier to find and upgrade the only instance of the factory. After the factory is upgraded, new instances of the new versions can be created on the fly. The factory is required to keep track of the object instances that it created. The old version of the factory transfers the old versions of the objects to the new factory, which can then upgrade every existing instance, thus finishing the upgrade process.

There are three kinds of modifications that can be done, changes that only modify the body of a method, changes that modify the signature of the method or interface by introducing new methods, parameters or changing the return type of the method. The third modifies also the data structures. These are the most difficult changes as every instance of the data need to found and updated on the fly.

Even the existing instances cannot be upgraded if they are in use. Therefore the factory upgrades the instance only when the instance is in quiescent state i.e. not actively in use. The mechanism for figuring out when the instance is in quiescent state is similar to Read-Copy-Update (RCU) used in the normal Linux kernel. Because it is possible that all the instances cannot be upgraded immediately it is imperative that there is a method for allowing the old instances to be used even through the new factory. Part of the factory implementation is

[4]The implementations were done in the Linux systems.

[5]These are really not threads but more like execution traces inside the OS. I use the the term thread because I have no better word.

Read-Copy-Update

RCU is a lock free synchronization method. Indirection is required for it to work. Any one is allowed to read the structure any time they want. Anyone wanting to write into it, has to first create a copy of the structure. The writer uses the copy for updates and when it is finished, it changes the pointer to point to the modified version. Whoever wants to read the structure after that sees the new versions and anyone who acquired the pointer before the pointer changed sees the old version. The old versions need to be removed some time, for this reason it is forbidden for the any thread using the data structure to block. When every CPU has been scheduled at least once there can be no thread using the old version and the old version can be removed.

a mapping function that is used if the object is still running the old version. It transforms the parameters of the new invocation to the parameters that would have been used in the old invocation. As a result the clients can use the new invocations and the factory makes sure that old versions see correct parameters without any effort from the clients part.

IV. Conclusions and future work

There are different ways to solve the problem of dynamic update of an application. Unfortunately some of the most basic problems have still not been solved. For example different versioning schemes provide absolutely no real information about the actual services or interfaces that the components or modules provide. A more fine grained approach would be valuable. Only few systems can even handle the simultaneous existence of different versions of the same component. There is no good solution that would ensure that the behavior of the new version is the same as the old version, unless the code has been formally specified (which it usually is not).

The situation is easier if we have redundancy to help in the process. Then we can simply bring a node down and upgrade it. Then bring the node back up. However this process is error prone and would benefit from automation. There is some work in this area but the solutions are not quite finished yet.

One solution is to create methods for creating upgradeable applications in the C language. That solution is an interesting even though it may not be exactly required. If we simply cannot tolerate downtime, then we must have redundancy, and if we can tolerate small amounts of downtime, we can use simpler approaches. In any case the current methods require still some manual intervention from the programmer.

References

[1] A. Stuckenholz, *Component Evolution and Versioning State of the Art*, ACM SIGSOFT Software Engineering Notes, Vol. 30 issue 1, Jan. 2005

[2] Miles Barr and Susan Eisenbach, *Safe Upgrading without Restarting*, Proc. of the ICSM'03, pg.129-137, Sep. 2003

[3] Haibo Chen et. al, *Live Updating Operating Systems using Virtualization*, proc. Virtual Execution Environments, VEE'06, pg. 35-44, Jun. 2006

[4] Baumann et. al, *Providing Dynamic Update in an Operating System*, Proc. of the USENIX '05, pg. 279-291, Apr. 05

[5] Marcin Solarski and Hein Meling, *Towards Upgrading Actively Replicated Servers on-the-fly*, Proc. of COMPSAC'02, pg. 1038-1043, Aug. 2002

[6] Stephen McCamant and Michael D. Ernst, *Predicting Problems Caused by Component Upgrades*, Proc. FSE'03, pg. 287-296, Sep. 2003

[7] Michael Hicks, Jonathan T. Moore and Scott Nettles, *Dynamic Software Updating*, Proc. of PLDI 2001, Vol. 27 issue 6,Jun. 2001

[8] Iulian Neamtiu et. al, *Practical Dynamic Software Updating for C*, Proc. PLDI 2006, pg. 72-83, Jun. 2006

[9] Michael D. Ernst et. al, *The Daikon system for dynamic detection of likely invariants* , submitted to Science of Computer Programming in June 2006.