Towards a Pragmatic Atomic Dynamic Software Upgrade System

Luís Pina
INESC-ID/Technical University of Lisbon
E-mail: luis.pina@ist.utl.pt

Abstract

The upgrade of a running program is often a disruptive operation that involves stopping and restarting the program's execution, becoming, thus, a serious problem for dependable systems. Yet, software upgrades are unavoidable. Unfortunately, current solutions for dynamic software upgrades are either incomplete or not practical.

In this paper, I layout the foundations for a new dynamic software upgrade system that provides atomic upgrades and is designed to integrate seamlessly with the current software development practices. This new upgrade system leverages on a software transactional memory to ensure that all the requests of an upgraded system are processed in a consistent state of the program, either before or after the upgrade.

1. Introduction

Several computer systems nowadays are required to run continuously without any interruption. The dependability of such systems is very important to avoid significant financial losses or even risk to human life.

The software running on these systems must be able to be upgraded to repair bugs, improve performance, or add new functionality. However, upgrading a computer system is a highly disruptive process. In the simplest case, it may involve shutting down the system and restarting it, thereby resulting in some downtime. This is unacceptable for most dependable systems.

A system that supports dynamic software upgrades, on the other hand, is able to upgrade its software without stopping the provided service, thus without any noticeable downtime.

For instance, techniques such as *rolling upgrades* and *big flip* [5] take advantage of existing redundant hardware (used for fault tolerance and load balancing) to enable the dynamic upgrade of a system. But, such systems also keep a state that is tightly coupled with the behavior and that must be converted to be compatible with a new behavior when the system is upgraded. Unfortunately, These techniques fail to support dynamic upgrades in an atomic way. With an atomic upgrade semantics, all operations submitted to the system are performed in the previous version or in the new one, never in an inconsistent intermediate version, a by-product of a still-in-progress upgrade process.

The upgrade system that I propose enables the atomic dynamic upgrade of a system. It also provides tools for converting the state across the different versions.

The approach that I propose for developing upgrades is designed to be practical, so that its interference with the development process is minimal. Moreover, it strives to reduce the complexity of the code that the developer is required to write to convert state: that code needs to deal only with two consecutive versions of the system.

In the following section I make an overview of the solution that I propose. In Section 3, I explain some of the goals that the proposed system must achieve, provide more details about the proposed solution and discuss some implementation issues. In section 4, I discuss previous work related to the proposed system. Finally, in Section 5, I conclude and discuss future work.

2. Conceptual Model

For the purposes of the upgrade system, an application consists of two layers: the **execution platform** that supports the execution of the application's code and the **upgradeable code**, that contains the application itself. The upgrade system is a part of the execution platform. The execution platform is not upgradeable.

The main components of an upgradeable application are shown in Figure 1. The execution environment runs the application by executing the **program**—a sequence of instructions that defines the behavior of the application. It receives a **stimulus** from the exterior and reacts accordingly, executing the program and sending the result—a **response**.

This behavior of the execution environment is defined by the execution platform.

Given the tight coupling of the program with the state that it manipulates, an upgrade may need to transform the program state to allow the correct operation of the new program. This transformation is done by a **transform function**: a function that initializes the new program state based on the current.

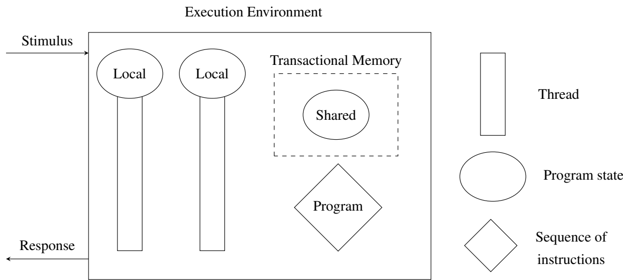

Figure 1: Conceptual model

2.1. Execution Environment

The execution environment keeps the current state of the application—the **program state**. To process each stimulus, it spawns a new thread that will access the program state to compute the response to the stimulus.

Although each thread has its own private data—the **local program state**—in which it keeps local variables and local copies of portions of the program state, all threads share some portion of the program state—the **shared program state**.

Since there is a portion of the program state shared by several threads, we need a synchronization mechanism to avoid concurrency related problems and resulting data corruption. In this work, I assume that a transactional memory is used—a **Software Transactional Memory** (**STM**)—to keep the shared program state correct. Each thread executes a request on its own transaction. The STM ensures that each transaction always has a coherent vision of the shared program state and that the shared program state is written atomically only if the transaction commits successfully. The local program state of each thread does not require any synchronization mechanism because it is private to that thread. The STM is also a part of the execution platform.

2.2. Software Upgrades

An **upgrade** is a special kind of stimulus. It contains a new program to replace the program currently in execution. Like all other stimuli, the upgrade is handled by a thread and must be perceived by the application as an atomic action.

Unlike other requests, however, the handling of an upgrade means that a new version of the program must be created. So far, the program has been considered read-only (immutable), thus each thread could read it without any synchronization mechanism. Introducing dynamic upgrades means that the thread's accesses to the program must be synchronized also, just like the accesses to the program state. Different threads may be executing different program versions just as different threads execute over different program states.

3. Proposed Solution

The goal of my work is to develop an upgrade system that: llows any part of the application to be upgraded at any time without disruption, provides an atomic upgrade semantics and integrates seamlessly with the current software development practices.

3.1. Development Process

Typically, software developers use a revision control system to keep track of the source code evolution. There are two different notions of version that I must distinguish. For the revision control system, a version is some piece of source code that was submitted later than the previous one and differs from it in some way. Such notion of version is commonly known as a **revision**. For instance, the developer starts by writing "Version 1" of the class $Point$ shown in Figure 2. This is **revision 1**. After that, he changes from polar coordinates to cartesian coordinates, making **revision 2**. Finally, he writes the transform function $convert$ and creates **revision 3**.

On the other hand, for the upgrade system, an **upgradeable version** is a portion of executable code that redefines some subset of the code currently in execution. Besides the definition of the new application behavior, an upgradeable version also has transform functions that allow it to convert the program state used by the immediately previous upgradeable version.

How do upgradeable versions relate to revisions? Not every revision can be an upgradeable version, since revisions are free to contain inconsistent code. But the developer can mark a revision as being an upgradeable version. An upgradeable version is a revision that must have the following two properties. First, it must be complete—that is, if the interface exported by some class C changes, all of C's client classes must also have a new version. Second, every transform function must be able to fully initialize an instance of the new version of any class present in the upgrade, given an existing instance of the previous version. Only revision 1 and 3 of class $Point$ are upgradeable versions. Revision 2 lacks the transform function.

On marking an upgradeable version, the upgrade system removes the transform functions and creates a new revision on the revision control system. To write a new upgradeable version of the system, the developer starts from this latter revision and writes the modifications while adding conversion code to the transform functions.

3.2. Composing Upgrades

The system evolves between upgradeable versions. Each upgradeable version is able to convert the state that exists in the immediately previous upgradeable version. If an instance is left in a version older than the immediately previous upgradeable version, the upgrade system will convert it successively until its version reaches the current upgradeable version.

If an upgradeable version is not put into execution, it must be submitted to the upgrade system when deploying

```
class Point { //Revision 1
  private double rho, theta;
  public double getDistance() {...}
}
class Point$1 { //Version 1 in Runtime:
  public static double $rho(Point$1 p) {...}
  public static double $theta(Point$1 p) {...}
  private double rho, theta;
  public double getDistance() {...}
}
class Point { //Revision 2
  private double x, y;
  public double getDistance() {...}
  //Introduced in Revision 3
  static void convert(old.Point o, Point n) {
    n.x = o.rho * cos(o.theta);
    n.y = o.rho * sin(o.theta);
  }
}
class Point$2 { //Version 2 in Runtime:
  public static double $x(Point$2 p);
  public static double $y(Point$2 p);
  private double x, y;
  public double getDistance();
  static void convert (Point$1 o, Point$2 n) {
    n.x = $rho(o) * cos($theta(o));
    n.y = $rho(o) * sin($theta(o));
  }
}
```

Figure 2: Evolution of a class, in source code (revisions) and runtime (versions)

a later upgradeable version.

3.3. Referring to the Old State

When writing a transform function, the developer must be able to refer to the previous version of the instance that he is converting. The proposed upgrade system enables this by generating a "dump" of the classes present in the previous version. These **old classes** have all the fields and methods marked as public, to allow the developer to access the internal state of the old instance. For instance, in reversion 2, the $Point$ class refers to a class $old.Point$.

The old classes are useful for the development process, to make the new version compilable. When deploying the upgrade, these classes are discarded and all references made to them are rewritten before the new version is put into execution. This process is illustrated in Figure 2. When comparing the source code that the developer writes (revision 2) with the code generated by the upgrade system (version 2), we find that the $convert$ method previously expected an $old.Point$ as first argument but it was rewritten to expect a $Point\$1$ in runtime.

3.4. Bytecode Rewriting

As shown in section 3.3, an upgradeable version is not yet ready to be put into execution after being compiled. Using a bytecode manipulation framework (such as Javassist [8], ASM [6] or BCEL [9]), the bytecode generated by the compiler must suffer several modifications, to solve the following problems:

1. How can several versions of the same class be present at the same time in the same JVM?
2. How can the upgrade system maintain the identity of the instances when converts them to new versions?
3. How can the new version be able to access instances of the previous version?

The first item is solved by simply changing the name of all the classes under the control of the upgrade system. The same class in different versions is actually a different class inside the JVM. Since the upgrade system must rewrite all classes, it is able to generate non-conflicting names for each class in each version. In Figure 2 there is a suggestion of non-conflicting names.

This process is not totally transparent: when debugging the application, the developer may see the generated names for the classes. Since the generated class names reflect the original names plus the version in execution when they were deployed, the new class names can actually help the developers throughout the debugging process.

Since the same class in different versions is actually a different class inside the JVM, the second item becomes relevant. In the solution that I propose, all references to instances of upgradeable classes are replaced by references to VBoxes, which are transactional locations used by the JVSTM [7] to keep the several versions of an object. Using VBoxes provides an useful extra level of indirection: when an instance needs to be converted, a new instance is allocated, initialized and added to the version list of the VBox that represents that instance. Thus, the identity of the instance is preserved. To trigger the conversion of an object, VBoxes can be modified to detect if there is a newer version of the class and convert that instance by calling the appropriate method. Finally, by using VBoxes, requests that happened before the upgrade continue to access the program state as it was before any part is converted, thus providing an atomic semantics to the upgrade process. The upgrade system rewrites the code generated by the compiler to introduce VBoxes when accessing upgradeable classes, preserving the type system of the application.

Finally, when converting instances between versions, the developer has full access to the old instance. This includes private fields and methods. To enable this, the upgrade system generates static methods that allow the access to such fields and methods. In Figure 2, we can see such methods: $\$rho$ and $\$theta$, for instance. All references to private fields and methods are rewritten, becoming references to these **accessors**. In Figure 2, the $convert$ method was rewritten to use the generated accessors. Besides this transformation, the upgrade system also removes references to the old classes, used for the compilation process, as explained in section 3.3.

4. Related Work

Some programming languages, such as Common Lisp [11], allow the developer to redefine parts of a running program, thus upgrading it. In the Java world, however, traditionally no such redefinitions were possible, even though some limited class redefinitions became possible with the Java Platform Debugger Architecture (JPDA) [1]. More recently, techniques such as binary code refactoring [10], extend the allowed redefinitions to a larger set. Yet, none of these provide any support for composing successive evolutions of a same class, allow developers to specify how objects should be migrated, or provide atomic semantics.

UpgradeJ [3] is a language level extension to the Java programming language to support dynamic upgrades. The developers have access to all the versions of any class by using explicit version numbers when referring to that class. Besides allowing the addition of new classes to a system, UpgradeJ supports two types of upgrades. The behavior of a class can be redefined and all of its instances immediately use that new behavior. When changing the structure of a class, the developer cannot delete fields/methods and he must create new instances to use the new class definition. One of the major problems with UpgradeJ is the burden that it places on the developer by requiring him to use explicit version control over every class plus decompose the upgrades into the few types of evolution that it supports.

The work that is more related to this work is the work of Boyapati et al [4]. They describe an upgrade system based on transform functions that provides good semantics that let the programmers reason about the transform functions locally. They consider a transactional multithreaded system very similar to what I describe in Section 2. They define modularity conditions that define the semantics of lazy atomic dynamic software upgrades. Such modularity conditions are the major contribution of their work to the proposed upgrade system. Nevertheless, there are several aspects in which the proposed upgrade system differs from mine. The authors argue that the upgrade system should take advantage of the encapsulation that results from good practices of object-oriented programming to meet the modularity conditions. The upgrade system that I propose uses a versioned STM (JVSTM [7]), and takes advantage of the versions of instances kept by it to meet the modularity conditions. This is a novel aspect of my work. It also describes in detail the implementation of an upgrade system designed for the Java language, dealing with several implementation issues such as, for instance, keeping the instance's identity across upgrades.

5. Conclusion and Future Work

I described the architecture of a practical system that enables the atomic dynamic upgrade of an application while integrating with the current common software development process.

Deploying upgrades and converting the application state appears to the rest of the application as an atomic action. The proposed upgrade system uses JVSTM [7], a versioned Software Transactional Memory, to achieve the atomic upgrade behavior.

Developing upgradeable applications using the proposed upgrade system has minimal interference with the common application development process due to its practical approach. The proposed upgrade system is designed to integrate with revision control systems, helping the developer to keep track of all application versions.

The developer is required to write a small portion of version aware code: the transform functions. These functions initialize the new state of the application based on the previous. To keep the writing of this code simple, the system was designed so that the developer has to deal with only two consecutive versions of the state.

The upgrade system that I propose in this document will be implemented on top of the JVSTM. After that, it will be integrated into the Fénix Framewok, which already uses the JVSTM, reaching all systems that use it, such as the FénixEDU project [2].

References

[1] Java(tm) platform debugger architecture. http://java.sun.com/javase/6/docs/technotes/guides/jpda/.

[2] Fénixedu. http://fenixedu.sourceforge.net, 2005.

[3] G. Bierman, M. Parkinson, and J. Noble. Upgradej: Incremental typechecking for class upgrades. In *ECOOP '08: Proceedings of the 22nd European conference on Object-Oriented Programming*, pages 235–259, Berlin, Heidelberg, 2008. Springer-Verlag.

[4] R. Boyapati, B. Liskov, L. Shrira, C. hue Moh, and S. Richman. Lazy modular upgrades in persistent object stores. In *In Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA*, pages 403–417, 2003.

[5] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, 2001.

[6] E. Bruneton, R. Lenglet, and T. Coupaye. Asm: A code manipulation tool to implement adaptable systems. In *In Adaptable and extensible component systems*, 2002.

[7] J. Cachopo and A. Rito-Silva. Versioned boxes as the basis for memory transactions. *Sci. Comput. Program.*, 63(2):172–185, 2006.

[8] S. Chiba. Load-time structural reflection in java. In *ECOOP '00: Proceedings of the European conference on Object-Oriented Programming*, pages 313–336. Springer-Verlag, 2000.

[9] M. Dahm and F. U. Berlin. Byte code engineering with the bcel api. Technical report, 2001.

[10] D. K. Kim and E. Tilevich. Overcoming jvm hotswap constraints via binary rewriting. In *In First ACM Workshop on Hot Topics in Software Upgrades (HotSWUp 2008)*, 2008.

[11] G. Steele. *Common Lisp the Language*. Digital Press, 2nd edition, June 1990.