Software Adaptation through Dynamic Updating

Emili Miedes, Josep M. Bernabéu-Aubán, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
Campus de Vera s/n, 46022 Valencia (Spain)

emiedes@iti.upv.es, bernabeu@upvnet.upv.es, fmunyoz@iti.upv.es

Technical Report ITI-SIDI-2012/010

Software Adaptation through Dynamic Updating

Emili Miedes, Josep M. Bernabéu-Aubán, Francesc D. Muñoz-Escoí

Instituto Universitario Mixto Tecnológico de Informática
Universitat Politècnica de València
Campus de Vera s/n, 46022 Valencia (Spain)

Technical Report ITI-SIDI-2012/010

e-mail: emiedes@iti.upv.es, bernabeu@upvnet.upv.es, fmunyoz@iti.upv.es

September 28, 2012

Abstract

Software systems evolve continuously. They need to be updated to fix bugs, to improve their performance or to adapt their functionality to new user requirements. In their first approaches, these updating techniques required that the corresponding program was not in use, substituting it with its new version and using such new version thereafter. This is still a valid procedure for single-user programs being run in personal computers. However, there are other kinds of software that should be always available, demanding a dynamic (i.e., at run-time) software update. This paper surveys the regular goals and currently existing mechanisms able to drive a dynamic software updating procedure.

1 Introduction

Software systems are continuously evolving. Some typical examples of software updates consist in changing the implementation of a given component, adding a new component, removing an existing one or fixing a bug or a security vulnerability.

When the software system being considered is a single application in a personal computer, there is no problem in updating that software. This only demands uninstalling the current application version and installing later a new one. Those actions should be done while the user is not interested in such software; i.e., while the application is not running. Such mechanism is also known as "*stop-and-restart*". This is not a constraint since those personal applications are not continuously in use. As a result, this kind of software update is "*static*"; i.e., the software being updated remains stopped while the update is done.

However, updates are not trivial when the software system is a service available to a potentially large set of users. In that case, stopping the service is not a valid option in the general case, and the software update should be "*dynamic*"; i.e., the update should be applied while the software system is still running with the aim of ensuring service availability. Note also that when a service update occurs, a logical contract among the service provider and its interested clients exists and should be maintained. Thus, the software non-functional requirements are commonly collected in a *service-level agreement* (SLA) [8] and the dynamic updating mechanisms should consider and respect those requirements.

Dynamic update mechanisms are useful for many types of software systems and applications. First, they are useful for upgrading the operating systems themselves [32]; i.e., to apply both the regular updates that fix bugs or include minor changes and the *major* upgrades that include a large number of changes, without forcing the user to restart the system. In a wider scale context, dynamic mechanisms are useful to update any type of web service or application that offers a continuous service to a large set of users. Without a dynamic update mechanism, to update such an application, a stop-and-restart model would be used. In that case, the ongoing user requests must be *aborted*, thus causing a significant nuisance to the connected users. Moreover, the application must be kept inactive during the time needed to perform the

upgrade and the corresponding testing, thus yielding it unavailable, which has a negative impact on the holder entity.

Another example in which a dynamic update mechanism is highly desirable is the *cloud computing ecosystem* as a general example of an on-line 24/7 high-scale environment. Indeed, one of the major features promised by any cloud computing provider is a high level of availability of the applications deployed *in the cloud*. Nevertheless, all the cloud providers run a software infrastructure or platform that sooner or later has to be updated and upgraded. As in the previous examples, a dynamic software update mechanism allows the cloud providers to update their systems while keeping the highest levels of availability and transparency from the point of view of the user.

The dynamic software update topic has been studied in the last three decades by many researchers, in different contexts, and a number of techniques and solutions of different types have been proposed. During that time, only a few surveys of dynamic update mechanisms can be found [73, 37, 42]. Nevertheless, to the best of our knowledge, no study surveying and classifying the common dynamic update techniques has been published yet.

The goal of this paper is to help the interested reader to understand some of the concepts and techniques found in the literature of dynamic software updating. First, Section 2 proposes a selection of requirements and goals that are *basic* in any dynamic software update mechanism. Then, Section 3 describes a number of techniques used in the existing literature. Section 4 discusses a number of issues related to some of the topics covered in the preceding sections. The paper is concluded in Section 5.

2 Requirements and Goals

As pointed out in Section 1, a dynamic update mechanism allows a software system to be updated while it is running. This means that to apply a change in the software, it is no longer necessary to stop the system and restart it once updated.

In the existing literature related to dynamic software updating, several authors provide their own *definition* of dynamic software update and list the requirements that a dynamic update mechanism may have. This section identifies a number of such requirements and goals. For each requirement, the main issues are described providing some references in which the topic is covered.

Continuity. *The update can be performed at run-time, without stopping and restarting the system to update and it does not interrupt the software execution*.

The first part of the requirement (the avoidance of a stop and restart) is the key attribute of the *dynamic software update* topic, as explained in Section 1. All the references that cover the dynamic software update implicitly assume it. Some of the authors that identify it explicitly are Fabry [28], Segal and Frieder [72, 29], Solarski [74], Murarka and Bellur [58] and Gregersen and Jørgersen [33].

The second part of the requirement can be seen as an *extension* of the first part. The goal is to ensure that the availability of the service offered by the software and its performance do not decrease significantly.

From a practical point of view, both parts of the requirement are needed to ensure that the software service is available. To show this, let us consider two *worst cases* that may happen in a *continuous* update mechanism. In the first case, the update process is dynamically applied but due to the overhead it imposes, it completely blocks the execution of the service thus yielding it unavailable. In the second case, the update process is also dynamically applied and can be executed in parallel with the service but reducing its performance to a minimum, which in fact would be a similar situation. In both cases, despite having the update process dynamically performed, the software is unavailable from a practical point of view.

On the other hand, the best case is such that the update process does not block the execution of the service at all (this is, it can be performed while regular service requests are being served) and it does not alter the service performance.

That best case is difficult to achieve so many authors consider a *relaxed version*. To this end, it is just required that the update process causes the *minimal performance overhead* or *disruption* to the updateable software, without specifying what the *disruption* may consist in ([50, 72, 29, 44, 17, 33, 63]). In other cases, this requirement is more specific, like in [28], which admits a *momentary delay* in the normal execution of

user requests or [74] which accepts that the update process may interrupt the application *the shortest time possible*.

Transparency. *The update process is transparent*, which means that it has no impact on its context (the user, the programmer and the managed application) beyond the results it provides (a software reconfiguration or upgrade). This is a manifold requirement, since several types of transparency can be considered:

- *User transparency*: The update mechanism is *hidden* to the user, this is, the user does not need to be aware of the update mechanism. These mechanisms do not require that users interact with the application in any specific manner nor have any specific knowledge or skills. Without this kind of transparency the user needs to know about the update mechanism and this will restrict the way users interact with the software.
- *Application transparency*: The update mechanism does not impose any constraint to the program about how it is designed or implemented, does not change the expected behavior of the program, does not impose any noticeable performance impact nor any constraint and is not noticeable to those system parts that are not related to it. Typical examples of constraints that may be imposed to the updateable components of an application are the use of specific programming or configuration languages, interfaces or base classes and libraries to include in the application. Application transparency directly implies programmer transparency; i.e., the update procedure does not require that programmers have any specific knowledge about the update process. So, programmers do not change the way they design and develop their programs.

Regarding the literature, these transparency requirements are identified by several authors. The *user transparency* requirement is only explicitly mentioned by Fabry [28] although it is implicitly assumed in every paper. On the other hand, Gregersen and Jørgersen [33], Solarski [74] and Bannò *et al.* [6] require *application transparency*.

Generality. *The update process is general.* This requirement actually has a twofold interpretation. First, *the update mechanism allows to apply different types of updates* of different degrees of complexity. The types of changes that are easier to apply are reimplementing some part of the system yet keeping the interfaces and the semantics intact and extending the software in a *constructive* manner (this is, keeping the existing components and adding new ones). More complex changes are modifying the interface of some of the components in an *incompatible* way or extending/replacing/exchanging the protocols that intercommunicate different system components or removing some existing components. A dynamic update mechanism that offers *generality* may allow any type of change that could be applied by the *classic stop-and-restart* update mechanism referred to in Section 1.

A second interpretation is that *the updateable systems can be of different types*. It refers to the ability of the dynamic update mechanisms to update *heterogeneous* components (those using different technologies, models, programming paradigms and languages, etc.) and this implies that there exists a system-independent update method that may be easily adapted to each specific component allowing whole system updates.

The first interpretation is used by Ajmani [1, 2] and Gregersen and Jørgersen [33] while the second is used by Solarski [74]. Panzica [63] refines the generality requirement specifying four different kinds of dynamic updates (ideally, any dynamic updating support should be able to manage all these update kinds) onto a software component: (a) *perfective update* (when the new component maintains the functionality of the previous one and extends its interface with some new operations), (b) *corrective update* (the component interface is not updated but its internal logic is modified: bug fixes, performance enhancements, etc), (c) *partial compatibility* (the component interface is partially modified; i.e., some of its operations disappear and are replaced by others), and (d) *incompatibility* (the component interface is completely changed). Hicks [42] refers to this requirement as *flexibility*, encompassing the two interpretations discussed above.

Robustness. *The functionality of the application being updated should be guaranteed.* This implies that the request being received by that application should always return a correct reply and generate a consistent (i.e., conformant with its specification) state; i.e., state corruption should be avoided. Unfortunately,

update correctness is an undecidable issue [37]. Besides corruption avoidance, some additional aspects of robustness may be considered in an updating mechanism (Hicks [42] describes some others: completeness, well-timedness, and simplicity):

- **Consistency preservation.** Consistency preservation means that the state obtained after a dynamic update should be identical to that obtained through a stop-and-restart (static) update. This implies that the client requests being served in the updating phases should always terminate (completion). Moreover, those requests should modify the application state and get answers both complying with the application specification and expected functionality (conformance). For instance, Kramer and Magee [50], Sridhar *et al.* [76], Solarski [74], Murarka and Bellur [58] require that the update process leaves the system in a *consistent* state but do not elaborate too much about the concept of *consistency*.

  Gregersen and Jørgersen [33] are a bit more specific and require that the software state after a dynamic update must be the same than the obtained by starting and running the application once the updates have been applied *statically*. The behavior is expected to be correct even during the update. Bannò *et al.* [6] require *data consistency* and also *consistency of flow* (the proper termination of pending requests). Finally, Panzica [63] identifies both variants of *consistency* pointed out above (i.e., completion and conformance).

  This consistency preservation is particularly hard to achieve when the new software version presents some incompatibility with the previous one. In non-distributed applications this happens when different software versions use different sets of data as their state, without any derivation rules that allow an immediate translation from the old version to the new one. On the other hand, distributed applications face a similar problem when software versions rely on different (and incompatible) sets of protocols that rule the interaction among their processes. In both cases some transitional interval should be defined, requiring the usage of some translation mechanisms while it lasts, complicating the software update.

- **Safety.** According to [42], safety means that "*malformed or otherwise incorrect updates should not cause the running system to crash*". This implies that the regular software engineering techniques should be used in order to guarantee the correct functionality of each of the modules being installed in an update, and a careful testing and deployment plan should be considered for managing the integration of those modules with the rest of the software that will host them.

- **Possibility of Update Abortion (Rollbackness).** *The update mechanism should allow the interruption of an ongoing update, rolling back all its effects.* In case of a single centralized application an update interruption is not a complicated action. Simply, the previous existing software is maintained and none of its modules is replaced. Since this only affects a single machine and a few modules, rolling back the update only requires a short time. Note that it is assumed that the update was still on-going and this implies that both the old and new modules were both installed in the system. So, it is simply a decision about which of the two alternatives should be maintained in the system and which one should be finally stopped and removed from that system.

  On the other hand, in a distributed deployment several application components need to be exchanged by their new versions in order to update that application. To this end some updating protocol is needed. The complexity of the rolling back decision depends on the amount of implied components and the progress achieved in the updating process. Note that the rollback may be required due to some fault in any of the components or in the updating protocol. As a result, the completion of the update would lead to a corruption of the application state or functionality and this should be avoided. This recommends that updating protocols forecast the actions to be taken in case of an update bug or failure, maintaining all resources needed for aborting any on-going update.

  In Solarski [74], the update is considered an atomic operation that is either successful or rollbacked to the previous version. In the update mechanism in POLUS by Chen *et al.* [17, 18], besides updating a component to the next version it is also possible to apply a *reverse update* to go back to a previous version, which actually is an effective rollback mechanism. Gregersen and Jørgersen [33] also consider the ability to *rollback* an update to restore the previous version of the software.

3 Updating Techniques

Different complementary techniques and mechanisms exist in order to comply with all the requirements described in the previous section. Some techniques are specifically designed for non-distributed applications consisting of a single program; others are intended for fault-tolerant distributed systems based on component redundancy. Let us describe this ample set of techniques analyzing the goals that may be attained using each of them.

3.1 Use of Indirection

There are a large number of authors that propose dynamic update procedures, mechanisms and tools based on the use of different sorts of proxies, intermediary objects and other indirection levels. All these mechanisms share the characteristic of using some indirection level allowing the installation of a new module and its proper configuration before replacing its old version. In order to complete such software update, the "*pointer*" that sets such level of indirection exchanges its value, referring then to the latest version of the module. These techniques are useful in both traditional non-distributed systems and in distributed systems based on a client-server interaction model.

The basic idea consists in adding an intermediary level between a caller and the dynamically updateable callee module it is accessing. Instead of having the caller directly access to the functions and procedures implemented by the callee module, it uses some intermediary code that points to the current implementation of that module. The state maintained by such an intermediary software piece can be overwritten at runtime.

Regarding the requirements outlined in Section 2, indirection is able to ensure a sufficient degree of *transparency* since a caller may not see any difference between the previous and the current versions of a called module, being thus unaware of the updates already completed in a given application. *Continuity* may be also ensured if the old and new module versions may coexist in the system before the update is started and the exchanging step does not force to block any call. On the other hand, *generality* and *robustness* are not directly granted by a mechanism based on indirection but are not impeded by it. So, other mechanisms should be considered in order to comply with those requirements.

Indirection was inspired by *dynamic linking* [21]. The latter allows the late binding of those modules placed in shared libraries, completing such binding at runtime. This permits that the modules being implemented by such shared libraries were updated, generating new library versions that can be used by the interested programs without requiring their recompilation and relinking. However, this is not yet a valid mechanism for dynamic updating, since not all parts of the application might be updated using dynamic linking but only those held in the shared libraries. Additionally, once one of the routines stored in such library is required by the application, such routine is loaded and bound, preventing the application from updating such routine while it is in use. In a dynamic updating mechanism there should be no limit regarding when a software piece may be updated. As a result, dynamic linking is not a valid example of dynamic update. It should be enhanced. Despite this, the fact of storing some kind of "*pointer*" (to be replaced later with the address of the called routine) that in its first access triggers the library loading and the routine binding is an example of the usefulness of indirection mechanisms and a first step towards dynamic updating.

Indirection has been used by a number of authors in order to implement real dynamic updating mechanisms. For instance, Fabry [28] was one of the first to use it, in combination with two different kinds of binary-level overwriting. The DAS operating system [32] was based on a segmented memory architecture and used the segment base registers in order to implement segment replugging, thus replacing the old version of a given module with its new one. Such mechanism demands that the interfaces of both module versions were identical. This is one of the first examples of software upgrades applied to operating systems. Bloom [10] reuses the idea of redirecting the calls to the updateable code by *remapping* some handlers, in the context of Argus programs. Lee [52] uses indirect addressing in order to allow the upgrade of modules in the DYMOS system.

Segal and Frieder [72, 29] use *interprocedures* which are some intermediary routines that *redirect* the client invocations initially targeted to the *old version* procedures to their *new version* counterparts. The authors also use a *binding table* which holds *pointers* to the updateable procedures. These pointers are overwritten in run-time, as new versions of such procedures are installed.

In POLUS [17, 18], Chen *et al.* use an indirection level by inserting a jump instruction in an *old-version* function, to redirect the invocations to the new version.

A refined solution consists in using, as an intermediary level, proxy objects that *simulate* the real implementation of the target object. The idea is that the caller code does not call the objects that implement the service but uses intermediary proxies. These offer to the invoker the same interface than the original target objects and hide the complexity of the dynamic update. There are a number of authors that follow this approach [56, 33, 19]. For instance, Purtilo *et al.* [65, 64] propose the use of a software bus to connect software modules by means of *proxies* that are automatically compiled from an additional declarative specification provided by the programmer. The proxies and the bus itself intercept the conventional calls to the functions of the modules and implement the functionality related to the dynamic reconfiguration of the modules.

Sridhar [76] includes the use of some intermediary objects called *Service Facilities*. These objects encapsulate the objects that provide the real service and offer the clients a *logical reference* that can be used as the real service object. Thus, these objects handle all the requests made by the clients. These objects also include the necessary logic to perform the dynamic rebinding, using some well-known design patterns (like Strategy) and some facilities offered by common programming languages (at least, C++, Ada, Java and C#).

Ajmani *et al.* [2] use some intermediary objects called *simulation objects* that represent past and *future* versions of the updateable objects. These objects are offered to the client code as if they were the real service objects. Internally, the simulation objects can manage and redirect the invocations issued by the clients to the real objects that implement the service.

In their framework FREJA, Bannò *et al.* [6] also use some specific Java class loaders and some intermediary objects to control the execution of updateable components.

3.2 Rewriting of Binary Code

There are some proposals that use some sort of *rewriting* of the binary code of the programs and applications to update. Several techniques can be identified.

Binary Redirection. Basically, *binary redirection* means dynamically modifying the binary code that is executed by a process (this is, the code saved in the main memory of the computer and directly read by its processor) so one or several call instructions that point to some function are changed to point to some other place.

As shown below, this was one of the first techniques proposed for a dynamic update mechanism. Nevertheless, it has a number of disadvantages. To begin with, it is strongly dependent on the particular compiler and especially on the hardware architecture it is aimed to. This reduces its generality and transparency.

This technique is difficult to automate, since each update depends on the binary code of both the original and the new version of the program. Furthermore, some precautions must be carefully taken. For instance, before updating the binary code of a function or procedure, it must be ensured that it is not currently being executed. Otherwise, undesirable effects may be produced. This compromises its robustness.

One of the first authors to propose the use of *binary redirection* was Fabry [28]. As a base context, there is some *client code* that performs a call to a fragment of binary code that implements a given function. To update the function, a new fragment of binary code is loaded in memory. The problem to solve consists in making that the old call from the client program stops *pointing* to the old code and points to the new code.

Fabry proposes two different alternatives to perform such a redirection. Both are based on adding a level of indirection (see Section 3.1) and rewriting some low level binary instructions to update such indirection level. In the first alternative, the client program makes a first call to a specific position in memory in which a JMP-like instruction is placed. This JMP instruction is dynamically overwritten, so it points to the address of the new version of the function.

In the second solution, when an update is performed, the old position of memory with the old JMP-like instruction is discarded and a new one is allocated, pointing to the address of the new version of the function. Then, the binary code of the client is modified to call the new JMP instruction. Regarding to the

first solution, this second solution has the disadvantage that it is necessary to modify the binary code of the client program.

General Binary Rewriting. The binary redirection idea showed above is actually a particular case of the more general concept of *binary rewriting* that consists in rewriting any part of the program. Some examples may be changing the implementation of a function or even its list of parameter types.

The modifications are applied at a binary level, this is, modifying the binary executables or even modifying the code currently loaded in memory, as it is being executed. This general technique has the same disadvantages than the particular *binary redirection* showed above, derived from its low-level nature.

Buck and Hollingsworth [14] propose a platform-independent API for managing code rewriting, allowing dynamic updates; i.e., the addition or removal of portions of binary code while the program is running. In that paper, the updates are focused on facilitating the software execution profiling and debugging. Nonetheless, their mechanisms are applicable to any other purposes.

Hicks and Nettles [44] use some binary rewriting techniques to modify the service implementation, data types and the client code that accesses to the patched code. To update the code of the program (*i.e.* the implementation of the functions) the authors consider two approaches: *code relinking* and *reference indirection*. The first alternative consists in changing the function invocations made by *client code* to the current implementation of the functions, forcing them to *point* to the new implementations. The second alternative consists in adding an intermediary indirection level among the new implementation of a function and the invocations to it (see Section 3.1), arguing that it would be more expensive and more complex to implement. The alternative finally chosen was the first one.

To update the type definitions, they also consider two options: *replacement* and *renaming*. The first alternative consists in *replacing* the definition of a type with a new version, by means of some *binary rewriting* mechanism. The second alternative consists in adding a new type definition and patching the code to use it, also by means of *binary rewriting*. The authors choose the second alternative because they consider it is simpler and more portable.

To apply changes to the code and the type definitions, *dynamic patches* are used. Given a version of the program to update and its next version, some automated tool computes the *patches* to apply. Besides creating regular patches (like with the `diff` and `patch` UNIX commands), the transformation of the data is also considered. The programmer can define *transformation functions* (see Section 3.5) to apply to the data any transformation needed.

Chen *et al.* [17, 18] describe POLUS, a tool that offers support to dynamically update a software system. In order to update a running program from version v to $v + 1$, the operation of the proposed procedure is as follows. From the source code of both versions, a *patch* is generated and then compiled into a dynamic library, which is *injected* into the running binary code. For each function that changes in the new version, POLUS inserts a jump instruction to redirect the program flow to the new implementation of the function, which is provided by the patch (see Section 3.1 for other forms of level indirection).

The use of dynamic patches is inspired by Hicks and Nettles [44], although POLUS is distinguished by the possibility to *reverse* this procedure. Given the version $v + 1$ of a running program, it is possible to rollback it to version v by applying an *inverse patch*. In Section 3.9 other examples of rollback-enabled mechanisms are given.

Binary Rewriting in Java. Another particular case of binary rewriting is its application to Java programs. The idea is similar to the general rewriting technique showed above, but in this case the binary language and format are those defined by the Java Language and Virtual Machine Specifications [31, 53]. The modifications are typically expected to preserve the *Java binary compatibility* [31].

As in the previous cases, this technique also has the disadvantage of depending upon a binary level although in this case, it has a minor practical impact, since the Java language is widely supported by many operating systems and hardware platforms.

Several authors have studied the use of binary rewriting in Java programs. One of the first proposals is due to Malabarba *et al.* [54]. It is based on an extension of the Java class-loader, allowing the maintenance of multiple class versions in a single program, and the development of a new Java Virtual Machine. This class-loader extension still demands some amount of binary rewriting, in order to adapt the existing objects

to the new class interface. Moreover, since Java uses dynamic linking, changes in the method tables are also required, thus using some of the indirection approaches described in Section 3.1.

Milazzo *et al.* [56] propose the use of an intermediary layer that ideally should be independent of any particular version of the Java Virtual Machine and be usable with any Java application (see Section 3.1). This layer includes a new Java class-loader that uses some Java rewriting techniques to modify the Java bytecode at loading time. Moreover, new intermediary interfaces and objects are defined and created to intercept the regular method invocations and redirect them to the proper service implementation. The client bytecode is also rewritten to use the new interfaces.

Gregersen and Jørgensen [33] propose a mechanism to dynamically upgrade Java programs by successfully saving the *problem of the version barrier*. In short, the problem can be described as follows. One of the techniques to load new Java classes consists in creating new classloaders and using them to load the new classes. Nevertheless, this solution has the problem that the new classes are not easily accessible from code loaded by other classloaders (for instance, by a parent classloader). The mechanism proposed in [33] can save this barrier by using *proxies* that are defined dynamically. The idea is to build dynamic proxies for the updateable classes and let them to act as *intermediaries* among client classes and real service implementation classes. See Section 3.1 for other techniques based on adding some *indirection* level. They also need to manipulate the Java bytecode, in a number of ways to prepare both client and server code to use and be used by the update mechanism.

The update procedure also includes a *lazy state migration* that transfers the state from an *old* version of a component to a new version (also see Section 3.5). One of the most remarkable *peculiarities* of this proposal is that the update mechanism in general and the state transfer mechanism in particular are *triggered* lazily, on demand. When an update is requested, it is not immediately applied, but lazily. Moreover, the state is not immediately transferred. Instead, the state of each *object field* is transferred individually, when it is first accessed. Their proposal also allows the rollback of applied updates (see Section 3.9).

Bannò *et al.* [6] also use some rewriting techniques in their FREJA framework, to apply updates to the bytecode of Java classes (see Sections III.C and III.D of [6]). This framework is based on the use of specific classloaders, some (centralized) update *managers* and some intermediary objects that control the execution of updateable components (see Section 3.1).

Finally, many tools and libraries offer services related to bytecode manipulation (including run-time manipulations). For the Java programming language, there are many alternatives like ObjectWeb ASM [60, 13, 51], CGLIB [16], Javassist [20, 79], Apache Commons BCEL [4], Javeleon [34, 35], JRebel [85] and some others listed in [48].

3.3 Quiescence

A number of papers use some form of *quiescence*. The basic idea is that an update of a component of a program, from a given version to the next one, cannot be applied at any moment during the execution of the program. Instead, before updating the component, the update mechanism must ensure that the update does not interrupt any running processes (for instance, the invocation of a service). To this end, different authors ensure that the component to be updated reaches some *stable* state. Depending on the author, this stability requirement is given a different name and described in different ways. In the end, multiple mechanisms can enforce it. The aim of all these techniques is to ensure the robustness goal described in Section 2.

Search in the Execution Stack. Some mechanisms inspect the process execution stack in order to know if a given function (or procedure) is currently being executed. If no reference to the function is found, then it is not being called from the program and it is safe to dynamically update such function (by redirecting the calls as in Section 3.1, applying a binary patch as in Section 3.2, etc.).

This technique is usually part of some other update procedure. For instance, Gupta *et al.* [39] inspect the stack to know if a given routine can be updated. They also use it in [38] to perform its state transfer procedure (see Section 3.5 for additional information on state transfer). Segal and Frieder [72, 29] also inspect the stack in order to know whether or not the procedure to update is being executed.

The main disadvantage of this technique is that it strongly depends on the architecture of the underlying machine. This problem is tackled by Purtilo and Hofmeister [65, 46, 64]. They propose the use of an

abstract format to represent the *frames* of the execution stacks and implement that format as a part of their dynamic software update solution based on their POLYLITH software bus.

Reach of a Safe Point. Some techniques depend on the program to reach a specific point or state. Once the program has reached such a point, the update can be applied safely. The program is forced to stay idle in the *safe point* while the update procedure takes place. Once the update finishes, the execution can be resumed.

This idea is used by a number of authors. The DYMOS system [52] used one of the first dynamic upgrade mechanisms able to ensure safe points. To this end, the minimal exchangeable unit is a procedure or module. Modules can be composed by multiple procedures. The DYMOS run-time support ensures that modules are executed under mutual exclusion. So, in the easiest case, a safe point is reached when the current thread exits the module to be upgraded. However, this is only possible when the interface of the old and new module versions are the same. DYMOS allows interface evolution and in those cases the system starts further checking in order to guarantee that a safe point is reached. To this end, the programmer may specify a list of procedures that should not be in execution for applying the update.

In Gupta *et al.* [39], these safe points are called *control points* and are determined statically, from the source code of the previous and next versions of the program. When a dynamic update has to be performed, the program is forced to *transit* to a safe point and then, generate a signal. Then, the update takes place and once finished, the execution is resumed. The authors also propose an extended model to be used with structured programs in which the *unit of change* is the function or procedure. They argue about the difficulty to specify the safe points and then propose an *inverse* approach based on specifying some *selected* functions the control should not be in at the time of change (see Section 4 of [39]). When a dynamic update has to be applied, first some stack inspection is performed, as explained above, to check that the program is not currently executing any of those selected functions. Once checked, the update is performed.

Chen and Huang [19] use the same idea in the context of dynamic update of OSGi applications (also see Section 3.10.1). Before applying an update to an OSGI *bundle*, they lead it to a safe point and then proceed with the required state transfer and perform the update.

Giuffrida and Tanenbaum [30] also use a central *Update Manager* component that dialogs with the updateable components, that must be *update-aware*. When one of the components has to be updated, the *Update Manager* leads it to a particular state in which a state transfer state can be safely performed (also see Section 3.5 for additional issues related with state transfer).

Communication Quiescence. The original concept of *quiescence* was defined by Kramer and Magee [50] in the context of dynamic software update of distributed systems. Informally, a node is *quiescent* if it is not going to start a data exchange or attending any data exchange with any other node. The authors argue that to apply an update that affects some nodes, they must be in a quiescent state.

When a node has to be updated, it is forced to *passivate*, this is, to reach a state in which the node is not communicating (in short, it is not bound in a communication with any other node and it agrees not to start a new communication). Moreover, all the nodes in its *passive set* (this is, all nodes that may communicate with the given node) are also forced to reach such a passive state. Once a node and its passive set are passive, the given node can be safely updated. As pointed out in [50] this procedure requires the collaboration of the application[1].

On the other hand, the *quiescence* concept and especially its *blocking requirements* have been criticized by some authors. They argue that in a general case, to passivate a component, a number of components must be passivated before, thus blocking them. In the worst case, all application components would be passivated, leading the application to an unavailability state which is totally contrary to the essence of any dynamic software update mechanism, violating its *continuity* requirement.

For instance, Vandewoude *et al.* [82] argue that the *quiescence* concept in [50] is, in general, stricter than necessary. They propose the concept of *tranquility* as a more relaxed alternative and justify that it can be used as a *stable state* in a dynamic software updating process.

[1] See also Section 3.4 for some other forms of *intrusion* and *coupling* between and application and the underlying dynamic update mechanism.

To understand the differences between quiescence and tranquility, one must compare the formal definition of the *quiescent* and *tranquil* states, according to [50] and [82], respectively. A node is in a *quiescent* state if a) it is not currently engaged in a transaction that it initiated, b) it will not initiate new transactions, c) it is not currently engaged in servicing a transaction, and d) no transactions have been or will be initiated by other nodes that require service from the node. On the other hand, a node is in a *tranquil* state if it satisfies a) and b) from the previous *quiescent state* definition and moreover, c) it is not actively processing a request, and d) none of its adjacent nodes are engaged in a transaction in which it has both already participated and might still participate in the future.

First, there is a difference between the c) clauses of these definitions. According to [82], the c) clause of the *quiescence state* definition implies that a node may be either *actively processing a request* or *waiting for a new request in an already active connection*, buy only the first case is required by the c) clause of the *tranquil state* definition. In practice, this means that a node may have started a transaction but if it is not currently servicing a request, it is considered *tranquil* and then it can be dynamically updated.

Moreover, according to [82], the d) clause of the *quiescence state* definition implies that no node has started or is going to start a transaction in which the given node takes part. Nevertheless, the d) clause of the *tranquil state* definition is less restrictive. It is only required that no adjacent node has started a transaction in which the given node has taken part and might participate in the future. The main difference is that the definition of *tranquil state* does not consider those transactions in which the given node has not taken part yet, so the nodes that started them do not need to be *passivated*. In practice, this means that, according the definition of *tranquility*, the update of a node is a *less blocking* process.

Pause and Resume. Another technique used by some authors consists in pausing the reception of incoming requests, waiting until the pending ones finish, applying the update and then resuming the handling of incoming requests.

For this, some *intermediary* level is used that may be implemented in various forms (see Section 3.1 for other examples that use some kind of intermediary level). For instance, some sort of *central* update *manager* or *intermediary* proxies may intercept the user requests and, if needed, pause them and rely them once the update is finished.

This technique is used by Bannò *et al.* [6] in their FREJA framework. They use several types of intermediary Java objects. On one hand, there are some *infrastructure* objects that perform the update and other management tasks. On the other hand, there are wrapper objects that wrap the regular service objects. The wrappers capture the regular service invocations made by the clients. If no update is to be done, the invocations are just redirected to the real service objects. When an update is requested, one of the infrastructure objects asks for the corresponding wrappers to *stop* attending new invocations (but *queue* them) and wait until the pending invocations are finished. Once the update is performed (by means of some Java bytecode-level rewriting, see Section 3.2), the blocked wrappers are instructed to resume their regular operation.

Other References. This idea of *stable status* or *quiescence* appears in many other references: [10, 7, 45, 9, 68, 3, 81, 43]. It can also be applied in other settings more or less related to dynamic software update but somehow different from the work referenced above. For instance, Dmitriev [23] talks about the dynamic update of methods of Java classes and the support offered by the HotSpot Java Virtual Machine. The mechanism is still under development, but it already offers some limited dynamic update mechanism, to ease the development and debugging processes and accessible by means the *Java Debugger Wire Protocol* (JDWP). The mechanism requires the collaboration of the programmer, which must ensure *"that the execution will actually reach the point where there are no active old methods"*, which can be seen as some kind of *user-ensured quiescence*.

3.4 Intrusion and Cooperation

A number of authors identify the necessity or dependence on some level of *intrusion* by the update mechanism, thus making the managed programs and applications aware of the update mechanism. This clearly breaks the *application transparency* requirement described in Section 2 but facilitates the compliance with

the *robustness* goal. Its objective is to allow a managed application to cooperate with the update mechanism, thus simplifying the updating tasks. This *intrusion* can take different forms.

A first kind of *intrusion* consists in defining special functions or procedures in both the update mechanism and the application to update. The idea is that, on one hand, the application offers a number of functions to be called by the update mechanism to perform its tasks. An example of this kind of *intrusion* is the use of `getState`- and `setState`-like functions assumed by many state transfer mechanisms (see Section 3.5) to retrieve or set the state of an updateable component. On the other hand, the update mechanism offers to the managed application other functions it may also call, for instance, to inform that its state has been changed or that the last requested update has been successfully finished. The update mechanism proposed by Kramer and Magee [50] is one of the first works that follows this approach. The authors identify two different coupling relationships between the update mechanism and the managed application. First, the so called *update manager* needs to invoke functions offered by the application (for instance, to request a state change). On the other hand, the application needs to invoke functions offered by the *update manager* (for instance, to inform that its state has changed). After justifying the need of both intrusion levels, the authors argue about the need of defining some kind of *standard interface* to communicate the update mechanisms and the applications. Moreover, they argue that the application must be involved in another way: it has to *promise* that it will remain *passive* long enough for the update to be completed.

In [30], Giuffrida an Tanenbaum propose a dynamic update mechanism based on an *update manager* that also depends on a close cooperation with the updateable components. When a dynamic update is to be applied to one or several components, the manager asks them to reach a *controlled state* (actually, some sort of *quiescent* state – also see Section 3.3). When the components reach such a state, they notify the manager who waits for all the notifications and finally proceeds with the update.

A second type of *intrusion* is the generalization of the first one and occurs when the update mechanism forces the whole application to follow specific constraints like the adoption of a given architecture, design principles, hardware platforms, software environments, programming languages or any other set of rules or conventions that force the whole application to be built or behave in a specific manner. This category includes all the proposals of update mechanisms based on the OSGi platform (see Section 3.10.1)

A third type of *intrusion* consists in making the application to provide some sort of *meta-information* that may be used by the update mechanism. One example of this type is *marking* the code of the updateable applications. Some update mechanisms require that the user marks those parts of the application in which a dynamic update may be carried on safely. This is the case of the proposals by Frieder and Segal [29] and by Neamtiu *et al.* [59], that allow the programmer to identify *safe update points* in the source code, in which an update may be safely performed. On the contrary, others depend on the user marking those parts in which a dynamic update should *not* be applied. This approach is followed by Hicks and Nettles [44] who propose a mechanism that allows the programmer to *mark* places in the code that should not be interrupted by a dynamic update.

3.5 State Transfer and Transformation Functions

Several authors identify the need to perform some sort of *state transfer* between the current version of an updateable item (typically an object or component, but it may also be a function or procedure or even the whole program or application) and the next version, in order to preserve it when the update is applied.

These mechanisms use a variation of the idea proposed by Liskov and Herlihy [41]. The basic idea consists in defining two *accessor functions* like `getState` and `setState` to retrieve and set the state of a component. Before replacing a component, the `getState`-like function is called and some *serialized* representation of the state is obtained. This state may be *transformed* in some way (see below) and then transferred to the new version of the updateable item, by means of its `setState`-like function. The usage of an abstract or serialized data representation matches the *generality* requirement outlined in Section 2. Despite this, even with such serialized format several problems arise when the module versions require incompatible data sets; i.e., when the data being used by the new version cannot be directly derived from the elder data.

In his Ph.D. thesis, Bloom [10] identifies the need of transferring the *volatile* state managed by the part of the program to be replaced, to the new implementation. Purtilo *et al.* [65, 64] propose an abstract representation of the data kept by the (dynamically reconfigurable) modules and the use of functions to

retrieve and set the state of a module. This allows the *migration* of the state of a given version of a module to the next one, once updated. The use of the abstract format allows to get the state of a running module before updating it and then restore it back or even move it from a physical node that uses a given architecture to a different node that uses a different architecture. Some other systems [38, 75, 74, 76, 33, 19] have used state transfer techniques in their update mechanisms.

Bannò *et al.* [6] identify the need of the *consistency of the data* in a dynamic update and the data transfer from the current component to the updated one. As already outlined above, one of the problems that may appear when updating a component from a version to the next one is that the new version may have an *incompatible state format*. Several authors consider this problem and propose the use of some kind of *transformation functions* that adapt the state of a component from its previous version to the proper current format. These functions are typically provided by the programmer, like in [10, 29, 65, 44, 2, 77, 58, 19].

3.6 Multiple Version Coexistence

Version coexistence is the ability of a dynamic update system to allow different versions of an updateable component to concurrently exist, providing a regular service according to their specifications. The coexistence interval lasts either until all on-going invocations to the old code terminate or until the old module is explicitly removed from the system (e.g., by an external managing component). If a mechanism of this kind is implemented, the *robustness* of the updating system is enhanced since it is easier to rollback an on-going update action in case of problems or bugs as the previous software version is still running and the requests initially targeted to the new version may be forwarded to the old one. Additionally, the *continuity* requirement is also easily achieved since it will be trivial to migrate between different module versions since all of them will be available and ready to run.

On the other hand, the support needed to provide this coexistence may have a cost, from different points of view. First, some additional software able to maintain multiple components providing the same functionality has to be implemented, which means a significant effort. Then, it may have some other cost at run-time, imposing some performance overhead compared with an update mechanism without such a support. Thus, in many systems, the dynamic update mechanism ensures that the new version of a component will never *coexist* with an older version. Some of them ensure this behavior by asking the program (or at least, the component to be replaced) to reach some stable or quiescent state (see Section 3.3), performing the update and *uninstalling* the previous version or at least preventing both versions to run at the same time.

Nevertheless, there are some systems that support version coexistence. For instance, in the context of *dynamic updating* of functions and procedures, Segal and Frieder [72, 29], define *interprocedures*, which are some sort of intermediary procedures that delegate on the real implementations. These interprocedures may be called from *old* client code (this is, client code that only *knows* the old version of the updated procedure) or from *new* client code, thus providing the *illusion* that different versions of the same procedure coexist.

Ajmani *et al.* [2, 1] follow a similar approach, by defining *simulation objects* as *proxies* that wrap the real service objects. For a given service object, it is possible to define proxies that represent the *past* versions and even *future* versions and all of them can coexist and be called by different pieces of client code that may be in different update stages.

POLUS [17] and [18, Section 2.2] allow the coexistence of old and new versions of the same *code* as well as old and new representations of data structures, after an update is applied. Moreover, it ensures that *old (new) code is only allowed to operate on the old (new) data, respectively*.

Our group has developed different database replication middleware systems where both the replication protocols [69] and the group multicast facilities [55] have been developed as pluggable components. To this end, the middleware relies on different meta-protocols that allow the installation and coexistence of multiple components. So, these systems allow the coexistence of multiple replication protocols and multiple total-order multicast protocols, being the applications able to use the most adequate protocol at every time. With this kind of support, it is easy to migrate from a protocol to another or to upgrade the implementation of any of the protocols, providing an adaptive platform to develop highly-available distributed applications.

3.7 Scheduling

Distributed applications replicate their components in order to ensure their availability, since failure and replication transparencies are complementary aspects of the *distribution transparency* [47] aimed at this kind of systems. In this scope, a software update should be carefully planned. This implies that some *scheduling mechanism* is needed, setting different updating phases and the nodes and application components that will be updated in each phase. To this end, a sequence of partial updates is progressively applied to different replica subsets of a distributed software, guaranteeing the availability of the services being provided by that application. As a result, *scheduling* mechanisms are needed for complying with the *continuity* (users do not perceive any sofware unavailability), *user transparency* (users do not modify the way they access such software) and *robustness* (the software always complies with its specification and its usage is safe) requirements described in Section 2.

In a system with static software updating (i.e., based on a *stop, upgrade and restart* sequence) *scheduling* has traditionally referred to the proper selection and announcement of the unavailability interval that such upgrade will provoke. Such unavailability interval is notified in advance to the potential application users in order to advice them to request the software service at other intervals. In a dynamic upgrade context, as explained above, this unavailability interval is avoided but some scheduling is still needed to appropriately coordinate the upgrading steps.

An example of scheduling mechanisms is proposed by Ajmani *et al.* [2, 1] with the use of *scheduling functions*. These functions are provided by the programmer of the managed system and may be called by the dynamic update mechanism to decide when each node has to be updated with respect to the other nodes. They identify different *update patterns* (borrowed from [11]) that may be implemented as *scheduling functions*. A first alternative for these functions is a *fast reboot* update. It consists in updating all nodes at once and this should be avoided since it yields the software system completely unavailable during the time required by the update. Another option is a *big flip*, which consists in first updating half the nodes at once and then, the other half. This option requires some kind of *load balancer* able to redirect to the proper nodes the user requests issued during the update. A more flexible option is a *rolling upgrade*, which consists in updating only a few nodes at a time (thus needing several steps to update the whole set of nodes). The disadvantage of this option is that it requires that both the previous and the next version of the managed software were compatible since they will coexist.

3.8 Replication

Replication is not an updating mechanism *per se*, but a necessary technique for ensuring the availability of distributed applications, overcoming the failures of one or several of their components. In order to manage a replicated system, the application developer should consider some replication and recovery protocols in order to comply with a replication model [36]. The *primary-backup* [15] (or passive) and the *active* [71] are the two classical replication models.

In a passive replication model all the requests are directly served by a single replica: the *primary* one. This replica propagates the state updates generated by each served request to the *secondary* or *backup* replicas before returning a reply to the client. In case of failure of the primary replica, one of the backups is elected and promoted to primary. When any of the backups fails then a new backup should join the set of replicas, needing a state transfer from any of the other replicas.

In an active replication model all the requests are directly served by all replicas at once. In order to guarantee a strong consistency among the replicas, the requests should follow the same delivery order in all of them. To this end, an *atomic multicast protocol* [22] is used. If any of the replicas fails, the remaining ones continue providing the requested services. So, no unavailability interval is identified in this replication model (note that the passive model may suffer a short unavailability period if the selection and promotion steps cannot be completed immediately). When a failed replica is recovered, it should receive a state transfer from any other live replica.

In both models some kind of state transfer is needed to accept a new replica or to recover a previously failed one. This is the basis for their recovery protocols, that should synchronize such transfer with the management of incoming client requests; i.e., the new replica should be aware of which will be the first incoming request whose effects need to be applied onto the complete state transferred by the recovery

protocol. With these building blocks (component redundancy, recovery protocols based on state transfer), the design of a general dynamic updating mechanism gets rather easy. It consists in preparing new replicas with the new software version and progressively replacing each of the existing ones with those based on the new software version. As a result, the replication mechanism combines several basic mechanisms described previously:

- The *state transfer* described in Section 3.5 for propagating the current service state to each one of the replicas being added with the new software version, dealing with any change in the data format or data usage between both versions. This is commonly integrated in an extended recovery protocol.
- The *version coexistence* described in Section 3.6 in order to allow that replicas with different software versions cooperate in providing the expected service.
- The *scheduling* described in Section 3.7 in order to select and arrange an appropriate replacing order for exchanging the software versions in all replicas of the application.

Thus, Solarski and Meling [75] propose a procedure to dynamically update a distributed system that uses *active replication*. The procedure relies on a group communication system that offers a total order message delivery service and operates by iterating over the available replicas, shutting them off, updating them (in a *static* way) and restarting them. This work is later extended by Solarski [74] by adding a procedure applicable to systems that use *passive replication*, following the general guidelines outlined above.

Besides the software upgrading process, there are other axes that allow system adaptation in a distributed system that relies on software replication. The second one is controlling the *consistency* [57] among the replicated component states. To this end, the software platform may provide a meta-protocol allowing the exchange of the replication protocol, as described in [69] in the scope of replicated database systems.

Another proposal of this kind is described in Wang *et al.* [84], adapting the consistency degree according to the observed rates of read and write operations issued by clients. The proposed architecture organizes the nodes in three categories: a master node, a (typically small) set of first-level replicas known as *deputy nodes* and the rest of nodes, considered second-level replicas and known as *child nodes*. The degree of consistency depends on the set of nodes that may manage each kind of request and the procedure being followed to propagate the state updates through all the replicas.

Thus, a *strong consistency mode* may be achieved when write operations sent to any replica are forwarded to the master, which sequences them and propagates them to all the replicas. Read requests can be sent to any replica and are served immediately because all replicas are up-to-date. This mode is allowed when the rate of write requests is tiny.

There are three other consistency modes that define different ways in which each kind of node participates in the write and read requests. They relax the global system consistency, but allow a faster request management.

3.9 Rollbacks

There are different techniques supporting the *rollbackness* requirement discussed in Section 2. Their objective is either the interruption of an on-going updating action due to some critical and unforecast error or to undo a completed update that has introduced some kind of problem that discourages its adoption.

The basic rollback techniques are:

- *Reverse updating*. In stand-alone systems (i.e., non-distributed ones) that base their updating techniques on patching, the traditional mechanism for aborting updates consists in building "*reverse patches*". This means that the programmer should implement two different versions of the patch, reversing the effects of each other.
- *Checkpointing*. Rollback-recovery mechanisms for distributed systems were surveyed by Elnozahy *et al.* [25]. They are based on checkpointing; i.e., on saving periodically the state of the components in persistent storage. The objective of these rollback-recovery systems is to ensure fault-tolerant

executions of the distributed applications in case of component failures, rolling back the state of the affected components and re-initiating their execution from a safe point previously stored on a persistent medium, thus minimizing the amount of lost data in those failure situations.

Checkpointing may also provide an adequate basis for rolling back a software updating process, maintaining the application data needed to re-take the execution in case an updating action were not successful. To this end, a checkpointing-based software updating mechanism should collect a checkpoint just before every updating action is initiated. If the updating action is eventually rolled back, the application is rewound and its current state is set to that contained in the appropriate checkpoint, allowing that the application were restarted from that point. This also allows further updates from that safe state.

Considering concrete samples of these techniques, POLUS [17, 18] uses a mechanism based on the generation of *dynamic patches*, employing a reverse updating technique for implementing rollbackness. POLUS uses a carefully designed *indirection* mechanism that avoids multiple indirections. When a function is updated from version v to $v + 1$ by means of a *dynamic patch*, POLUS inserts a *jump* instruction to redirect the incoming calls to the proper version implementation. If the function is updated by succeeding requests, to versions $v + 2$, $v + 3$, etc. then POLUS makes the *jump* instructions to directly point to the latest version of the function, thus avoiding unnecessary redirections. This mechanism allows the rollback of several updates, one after another, so it is possible to rollback from version v to $v - 1$, then to $v - 2$, $v - 3$ and so on, as long as it is possible to build the proper reverse patches. Besides applying the patches, POLUS also *undoes* the insertion of the corresponding *jump* instructions, thus again avoiding unnecessary *back and forth* redirections.

Brown and Patterson [12] propose a checkpointing model for rollback mechanisms that solves the *external inconsistency* problem. This problem arises when the data being rolled back had already been read and processed by the users.

The proposed model is based on three stages or steps: *rewind*, *repair* and *replay*. In the *rewind* step, the rollback mechanism discards the changes to data made after applying the update. Previously, the rollback mechanism saves a *semantic representation* of those changes, so they can be re-applied later. In the *repair* step, the update is rollbacked. In the *replay* step, the saved changes are re-applied, but using now a fixed version of the application.

As an example (and proof of concept in their prototype), the authors test the rollback of updates in a regular email client application. In the *rewind* step, the changes are saved using a *semantic representation*. Instead of *logging* the changes made to the filesystem (e.g. the deletion of a file record, when deleting an email message), the rollback mechanism saves the *action* performed by the user, in an abstract way (e.g. "delete the message with id N"). This abstract action is re-applied in the *replay* step, once the update is rollbacked.

The proposed model presents a *lack of genericity* problem, since it depends on particular protocols (IMAP and SMTP, JDBC, XML and SOAP, etc.). It also depends on the possibility of expressing each possible user action in terms of the given protocol. For instance, the deletion of an email message can be represented in terms of a `DELETE` IMAP command but some other user actions (e.g. the creation of a new message draft) may not be IMAP-representable.

3.10 Other Techniques

Many techniques described up to now comply with the *generality* requirement. So, they are platform- and programming-language-independent, being usable in multiple scenarios. However, there are other mechanisms that do not comply with this generality principle but demand some attention since they provide a good software updating support in their specific target system.

Those proposals are described in the sequel.

3.10.1 OSGi

OSGi [62, 61] is a platform to build Java applications from a number of modular, reusable and collaborative components (called *bundles*), that can be dynamically reloaded. Each bundle is a Java class that

implements a specific interface (`BundleActivator`). It provides the two basic methods that define the *life cycle* of the bundle, `start` and `stop`, to start and stop the execution of the service offered by the bundle, respectively. Moreover, a bundle may implement additional interfaces. For instance, there is a `ServiceListener` interface to receive events related to the bundle (e.g., when it is registered or unregistered in the OSGi implementation).

Each bundle is packed in a *Java Archive* (JAR) file. This file includes a *manifest* in which the programmer specifies some metadata, including the version of the bundle and its main class (that implementing `BundleActivator`). The programmer also specifies the packages that the bundle *exports*. When a bundle is registered in an OSGi server, this knows which services are *offered* by the bundle. The programmer can also specify the packages that the bundle *imports*, by providing a list of them and optionally, for each package, the minimal version that is required. This expresses the dependencies the bundle relies on, including general packages from the OSGi standard API and other services provided by third parties.

An OSGi server (or implementation) acts as a software bus. Bundles are first registered in it and then started by it. Once started, a bundle may register a service under a given symbolic name. It may also get *references* to other services (provided by other bundles), looking them up by their symbolic names. This means that if a service depends on another service, it does not need to depend on a specific implementation of it. Instead, it can rely on any service registered as an implementation of the required service.

Nevertheless, one of the main strengths of OSGi is that it allows to dynamically reload bundles. Once a bundle is registered and started, its source code can be updated and recompiled and the new bundle version may be reloaded. Then OSGi keeps the old bundle version available to those bundles that already had a *reference* to it (in order to let them progress correctly) and offers the new bundle version to those bundles that get a reference to the bundle from that moment on.

OSGi offers two operation modes. In the *Bundled Application* mode there is an OSGi server that acts as a container for one or more OSGi applications. This model is similar to that of the Apache Tomcat application server acting as a container for a number of Java web applications. In the *Hosted Framework* mode, the OSGi implementation is *embedded* in a given application.

A short introduction to OSGi can be found in [80]. Moreover, there are a number of implementations of OSGi, like Apache Felix [5], Concierge [78, 67] (especially designed for resource-constrained devices), Equinox [24], KnopflerFish [49] and Oscar [40], among others.

Moreover, there are some other proposals that extend OSGi or are related to OSGi in some way. For instance, Rellermeyer *et al.* propose R-OSGi [66], an extension of the standard OSGi specification to build distributed systems. Another alternative also focused on distributed and cloud systems is OSGi*4C* [70]. Finally, Chen and Huang [19] propose a mechanism to dynamically update the bundles of an OSGi application.

3.10.2 Dynamic software update in the .NET platform

There are some dynamic software updating mechanisms integrated in the .NET platform. To begin with, the *Managed Extensibility Framework (MEF)* and the *Managed Add-In Framework* (MAF) allow dynamical code loading.

MEF presents some similarities with the OSGi framework. Both allow to build applications that can dynamically *load* add-in components (as plug-ins). Moreover, both have a declarative mechanism to express relationships among components. As in OSGi, each MEF component declares its dependencies (or *service imports*) and its capabilities (or *service exports*). When MEF loads a component it checks its service imports, decides if other already loaded components *export* those services and in such a case, *connects* them. Moreover, MEF also checks the service exports of the components and decides if they can be connected to the *service imports* of other already loaded components.

As in OSGi, the advantage of this model is that the applications do not need to *hardcode* their dependencies on other components. Instead these dependencies can be resolved in run-time, in a similar way as in an *Inversion of Control (IoC) container*. Moreover, the extension components are not bound to the .NET assembly [2] of the application they are *extending*, so they can be easily reused with other applications. The

[2] In the .NET platform, applications are organized in *assemblies*, composed by one or more types (classes, interfaces, etc.). An assembly is somehow *similar to* a Java package. Assemblies can be dynamically loaded although it is not possible to dynamically load a single type. Moreover, a .NET application has one or more *application domains*, which are the *isolated contexts* in which

main drawback of MEF from the point of view of dynamic software update is that it does not allow the *dynamic unloading* of the components, thus prohibiting any kind of *dynamic update*.

MAF is a technology similar to MEF. Regarding dynamic software updating, the most important difference is that MAF allows the application and the add-in components to be in different .NET *application domains* so the add-ins can be dynamically unloaded.

There are other attempts to define some sort of dynamic update mechanism in the .NET platform. For instance, in [27], Escoffier *et al.* discuss some issues related to dynamic update of .NET applications. Their goal is to design some sort of *OSGi for .NET*, although they identify some features of the .NET platform that prevent from getting the same semantics of OSGi. First, the load unit (and also the *unload unit* when it is possible) is the assembly, which typically contains a number of types (classes). To load and unload a single type, it must be the only type included in a single assembly, which is not practical. Second, dependencies among assemblies are included by the compiler in the executable binary code, which makes difficult to dynamically change them. In contrast, Java dependencies are resolved at run-time, which eases reusing the compiled Java classes. Third, the *load order* of the classes and assemblies is *strongly* controlled by the virtual machine and the user cannot change it easily. In contrast, Java applications can modify the class load process by using their own class loaders. Finally, the name of an assembly is part of the name of the types contained in it, which also makes difficult to reuse such types.

They propose a number of alternatives to dynamically load (and unload, in some cases) .NET types. A first alternative consists in using a single application domain and several assemblies loaded into it. One of these is special and loads the rest of them. As all the assemblies are in the same application domain, every class can invoke methods from any other class using regular local invocations. The main drawback is that such assemblies cannot be individually unloaded. To unload a type, the whole application domain should be unloaded, which in practice is equivalent to stop and restart the whole application.

A second alternative consists in using several application domains in the same application. Each assembly that needs to be unloaded is in its own application domain. To unload a type, which is contained in a given assembly, the corresponding application domain can be unloaded. As pointed out above, the drawback is that application domains cannot communicate by regular local invocations but some other IPC-like mechanism must be used, with the consequent performance penalty. Java applications that use different class loaders experience a similar issue, since a class loaded by a given class loader cannot access another class loaded with a class loader that belongs to a different *branch* of the class loader hierarchy, by means of regular local invocations.

A third alternative consists in using .NET's *shared domains*, to hold assemblies that may be common to all application domains. Other assemblies may be included in different application domains.

Two alternatives more (quite similar to each other) consist in having one specific application domain to hold assemblies that may be common to the rest of the application domains. In one case, the special application domain is a *.NET shared domain*. In the other case, the special application domain is a regular application domain that just has that special role. In both cases, both solutions offer a better performance when a class invokes methods from a class that belongs to the *special application domain*. On the other hand, both solutions suffer from the same performance penalty of the two first alternatives, when invoking classes that belong to other application domains.

Finally, they conclude by identifying the two main issues of .NET that prevent from designing an OSGi-like infrastructure for .NET. One is the *inability* of unloading individual assemblies from a given application domain. The other is the need to use slow IPC-like mechanisms to communicate classes belonging to different application domains.

3.10.3 Dynamic software update in Erlang

Erlang [26] is an interpreted concurrent functional programming language that can be used to build distributed fault-tolerant (and soft-real-time) applications. Erlang allows to dynamically replace single modules of an application.

Erlang allows a module to have two concurrent versions: the *current* version and the *old* version. When a module is first loaded, it is in its *current* version. It then can be replaced with a new instance of the

assemblies are run. Classes in different application domains cannot communicate directly (by means of a regular local invocation), but some inter-process communication (IPC) mechanism is needed (thus rising the cost of the invocation).

module. Then, the current version becomes the *old* version and the new instance becomes the new *current* version.

Erlang allows to keep both versions in execution. Once applied the update, the *current* version is generally used but old existing processes that were accessing the updated module go on working with the *old* version until they normally finish. If a new (third) version of the module is installed, then Erlang removes the old version and finishes the pending processes that were using it. Then, it installs the new version according to the procedure explained above.

Moreover, Erlang allows to define special functions that may be run when a module is loaded. Such functions may apply the needed state transformations.

4 Discussion

The *dynamic software update* topic has been studied for decades (the first well-known references are from the 70s), in the context of both stand-alone and distributed systems. Many papers have been published and multiple practical mechanisms exist. Some of them are for specific contexts and situations (as described in Section 3.10) but others were designed to be generally applied and used.

Still, any user of current software can note that these techniques are not being *universally* applied. The users of current software (for instance, in a *household* context) are used to restart their software applications and even the whole operating system when they have to update them. For instance, web browsers need to be stopped and restarted in order to be updated.

Operating systems usually have some sort of *update manager* to detect, download and dynamically apply updates to different components of the system and even its own *kernel*. In some cases, updates can be applied without having to restart the system. In other cases, some types of updates require a complete system restart. And in some other cases, updates are dynamically applied but still require a system restart for using them effectively (as it typically happens when an update is applied to *core* components of any operating system, for instance).

Regarding web applications and services, the current situation is diverse. On one hand there are a number of popular services (that usually belong to *large* companies), like web search engines, email services, social networks, storage and multimedia broadcast services, etc. that have their own mechanisms to dynamically update the applications. These applications use replication techniques that allow the updates to be transparently applied so the users do not realize when the updates happen (unless the update modifies the existing user interface).

On the other hand, there are other *small* web applications, typically belonging to *smaller* companies that do not have the same availability of resources of the *large companies*. In many cases, the offered services have to be temporarily shut down while the updates are applied. In these cases, the procedure followed by the service administrators consists in redirecting the user requests *to some other place* (for instance, to a static page that informs about the unavailability of the service), stopping the applications and servers to update, applying the updates to the programs and/or the data, checking the changes and restarting the servers and applications. During the time needed to perform these actions, the service is typically unavailable. Thus, users are forced to interrupt their use of the application and wait until the services are restored.

To avoid these drawbacks, software should have some dynamic update mechanisms allowing updates in run-time, in the most transparent possible way to the users. This would avoid the interruption (and thus unavailability) of the service and the corresponding nuisances to both users and service providers.

Cloud computing systems should also be considered. A transparent dynamic update mechanism makes even more sense in the applications that are executed in these systems, since they are potentially (and continuously) available to a larger set of users. For this, cloud systems must be dynamically updateable. Three different types of *cloud architectures* are typically identified with the names *Infrastructure as a Service*, *Platform as a Service* and *Software as a Service* [83]. In all these cases, there are a number of software components that may be updated from time to time. In case of IaaS providers, the infrastructures typically offer to the user abstractions as virtual machines with the *appearance* of a full operating system, so applications can *think* they are executed in dedicated servers. In some other IaaS cases, the user has some *virtualized environment* in which she may *deploy* applications. In case of PaaS providers, the offered

service consists in a programming platform where users build their applications. In case of SaaS providers, no infrastructure is offered but one or more *final-user* applications, that can be reached through any regular web browser and used like any other locally installed application. In all these cases there are multiple software components that will evolve, having user transparency, availability and quality of service as their common requirements to be satisfied. To this end, dynamic software updating techniques are a must. Additionally, the user applications that are executed onto IaaS and PaaS systems may also benefit from such update mechanisms, in order to provide a continuous service to their users. These mechanisms may exploit the *elastic* nature of the underlying cloud systems.

As we have showed, the use of dynamic update mechanisms is appropriate in many present types of software systems and applications. Nevertheless, not all the techniques referred to in Section 3 are equally appropriate to current software. In the rest of this section some related *issues* are discussed.

Issues related to *low-level* techniques. Several techniques depend upon low level of abstraction details. For instance, some techniques are based on the inspection of the execution program stack (see Section 3.3), to know if the function or procedure to update is currently being used. The programmer of a dynamic update mechanism based on such low abstraction level techniques must know which is the format of the memory space bound to processes, how the stack *frames* are stored and how data is stored in the frames.

Other techniques use several forms of *rewriting* at the binary level (see Section 3.2). In some cases, *binary rewriting* is used to *redirect* the execution of the code. The idea is to *install* the updated code and *redirect* to that code the calls to the old code, by modifying memory address pointers (this is, making them to point to some *other place*). Sooner or later, the old code will be no longer used and only its updated version will be used. This is the case, for instance, of the techniques based on *binary patches*. To apply this technique, some sort of pause has to be *imposed* in the execution of the program to update, for instance with any of the techniques discussed in Section 3.3. This pause may be brief so the impact on the availability of the program may also be small. In other cases, longer parts of the program are rewritten directly altering the instructions loaded in main memory. Nevertheless, in this case, the availability of the program may significantly decrease in case a large part of the program had to be rewritten.

Nowadays, both types of techniques seem undesirable, for a number of reasons. First, these techniques require a deep knowledge of very low level of abstraction details, related to hardware architectures (formats of the instruction set, memory addressing modes, etc.), which severely limits the portability of the techniques. Besides, we have to consider the current *trend* to use high level programming languages, most of which are interpreted and depend on some sort of interpreter or virtual machine: Java, C#, Perl, PHP, Python, Ruby, Erlang... The portability of the programs developed with those languages is based on the existence of interpreters and virtual machines that are available in different platforms and that ensure the same semantic behavior. Regarding the Java and .NET platforms, there exist formal specifications of their intermediate languages (the Java Bytecode and the Microsoft Intermediate Language), which means that the dynamic update solutions built with such platforms may use some binary rewriting techniques and still be portable. Even then, these solutions would depend on a specific version of the intermediate language. The developers would be forced to follow the evolution of the platform and the programming language (and specifically, the intermediate language) and adapt their updating mechanism to the changes they experienced, in order to have an update mechanism that could be used by current software. On the other hand, regarding many other languages for which no intermediate language exists, it does not seem an easy task to design a dynamic update mechanism that uses binary rewriting (or in general, any other low level technique) and at the same time was easily portable to different platforms.

For all these reasons, the use of binary rewriting techniques or other low level of abstraction mechanism is not a recommendable option when continuity and generality are two basic requirements.

Issues related to the use of indirection. The use of indirection has some advantages. Dynamic update mechanisms can use an intermediary level between a client program and a service it uses. This level may *wrap* the real implementation of the service. It may capture the invocations to the service and manage them as necessary, by blocking or pausing them, relying them, modifying them, etc. Such an intermediary level could also be used to perform other auxiliary tasks like managing authorization issues, statistically accounting the use of the service, logging and monitoring it, etc.

Moreover, in client/server systems, when used along other techniques (as discussed later), it eases the application of the updates. For instance, it allows the *coexistence* of versions of the same program.

Nevertheless, the use of indirection also poses some disadvantages. First, the use of an intermediary level always imposes some overhead in run-time. If the update mechanisms use intermediary functions, *interprocedures*, proxy objects, etc. each invocation of a *service* function or method first goes through such intermediary code, which imposes a run-time overhead. This overhead may be small and even negligible, especially when compared against the benefit obtained by using it or it may be significant and non-negligible and yet the intermediary level may be considered useful.

Another issue is the impact that the use of such intermediary levels would have on the development of the user application (this is, on the *application transparency*). The use of a specific technique, a middleware library or any other artifact that provides or helps to build such an intermediary level may require some specific knowledge or skill to the programmer and may have an impact on the design of the applications or the development process. This should be considered when taking the decision of using some intermediary level artifact. Ideally, this should be transparently integrated into the application and required the minimal maintenance possible in order not to *disturb* the designers and developers and not keep them away of the core design and development processes.

Issues related to *version coexistence*. *Version coexistence* offers some advantages, especially when combined with other techniques and approaches (like the use of indirection). For instance, in a client/server context, it allows to update a software server that is being currently used, so the current clients can go on working normally with the *old* version of the server and, at the same time, those clients *connected* to the server once updated can directly start working with the newer version. Otherwise, without version coexistence, once updated the server, the existing clients would be forced to either halt or be updated, which may cause significant nuisances to the users.

One of the technologies that offer some *implementation* of this technique is the OSGi standard. In OSGi it is possible to have two different versions of the same service. When a *client* program has a *reference* of a service and this is updated, the client program is neither aborted nor forced to be updated. Instead, it keeps its *reference* to the *old-version* program and is able to go on working with it normally. If another program gets a reference of the same service, once updated, it is given a reference to the new implementations. Both versions *coexist* in the OSGi server and can be used normally. Finally, when all the references to the old version of the program are finally discarded (for instance, because all the programs that hold them are finished or simply stop using them), then the old version of the program is *unloaded* from the OSGi server and finally discarded.

Nevertheless, version coexistence poses some problems. For instance, it is necessary to think about the consistency of the data *shared* by the different versions of the program. If two different versions of the same program or component access the same data set, they should access the data in a *consistent* manner. In some cases, they may be allowed to access the same set of data, if the accesses are synchronized and the proper data transformations are applied. In other cases, it may be necessary to keep separated *snapshots* of the data, so each version of the program or component can access its own data set. In this later case, some *merge* mechanism may be needed to reconcile one snapshot to the other. Moreover, additional precautions may be taken, like *synchronizing* the access to specific resources like other data sources, physical resources, etc. In any case, the support to version coexistence leads to complications that must be evaluated.

Issues related to replication. As already explained in Section 3.8, replication takes as its basis component redundancy in order to ensure that the replicated software remains highly available and scalable. This allows even the usage of a traditional stop-and-restart (static) update mechanism for each replica, relying on the other live replicas that remain active serving client requests. When a replicated software needs to be updated, its upgrade can be arranged with the composition of several mechanisms described above, mainly: state transfer, version coexistence and scheduling. The way in which these mechanisms are driven depends on the replication model being used.

The passive model [15] uses a single primary replica that directly serves all requests, propagating later the effects (i.e., state updates) of such requests to the backup replicas. In order to improve its performance, relaxed passive models allow that read requests were served by any replica, distributing thus the read-

requesting workload of the primary replica among all the others. On the other hand, the active model [71] demands that all client requests are delivered to all replicas and served by each of them.

The passive model distinguishes two different replica roles: primary and backup. The backup role only needs to manage the reception and application of the state transfers initiated by the primary. As a result, software updating can be implemented starting with the backup replicas. To this end, each replica can be stopped and restarted with the new software version. Such update can be static since backup replicas do not serve client requests in a direct way, but only deal with the state transfers sent by the primary. The renewed replica may adopt a new identifier when it rejoins the system, requiring a complete state transfer in that recovery procedure. Thus, it does not matter how much time it has required to complete the static software update. Besides updating all backups to the new software version, the primary should be stopped and one of the updated backups promoted as a new primary, managing these steps as a regular primary failure event. The old primary is then statically updated and later restarted as a new backup replica.

A careful update scheduling is needed for deciding the order in which the replicas must be updated, ensuring the availability of the service if failures arise. Version coexistence (or state transformation) mechanisms should be also used since the recovery full state transfers should be understood and correctly managed by the renewed backup replicas. Additionally, since not all backup replicas are renewed at once, some of them will be able to execute the new software version while the primary still works with the old one.

In an active replication model [71] all replicas share a single role and no partial state transfer (propagating the generated updates) is needed in a regular request service. Despite this, the regular updating procedure is similar to the one described for the passive model, involving an adequate scheduling of the sequence of replicas to be updated and state transfer or transformation mechanisms able to deal with the version coexistence arising while there are two replica subsets executing different software versions. Those scheduling and state transformation mechanisms are the key for maintaining the consistency among replicas:

- When a replica with the new software version is restarted, assuming a static update, it should receive a full state transfer. If the sender of such state still executes the old version of the software, a transformation will be needed. As soon as there are other new-version replicas running, one of them may propagate the full state, avoiding the need of such transformation. This should be considered in the scheduling of the software update.

- If a static update model is assumed and some replicas already work with the new version of the software, the remaining old-version ones may discard the received client requests until they are restarted with the new software version. Note that such requests may be also answered by other new-version replicas and the reply returned by old-version ones might be different. So, such old-version replicas are logically stopped when any new-version replica is running. Again, this demands a careful planning when the sequence of replicas to be updated is scheduled.

- If a dynamic update model is used, all replicas will upgrade their software at once, replacing some of their modules. To this end, some low-level mechanisms are needed to implement the dynamic update, as described in Section 3. Besides this, a careful scheduling and synchronization should ensure that the update is applied in the same logical moment at every replica; i.e., that each replica has processed the same requests before the update and will process the same sequence after it, avoiding thus any state divergence among the replicas. That synchronization may be achieved multicasting an update start message using the same total-order channel that propagates the client requests to every replica.

As it has been shown in this few configuration examples (combining updating and replication models), the interaction among state transfer, state transformation, version coexistence and update scheduling mechanisms is the key for designing eficient software updating protocols for replicated software. The combinations mentioned above are only some of the possible ones, but further work is needed in order to find the best strategy for each kind of replicated application.

Modern distributed applications are deployed on cloud systems. These systems are adaptive and scalable; i.e., elastic. Replication is needed in that context as a basic mechanism for achieving scalability. Adaptiveness requires a fast system reconfiguration when the application requirements (workload, resource usage, economical costs,...) vary. This adaptation usually requires varying the current number of replicas

supporting each of the distributed application components. In these scenarios, *service level agreements* (SLA) set the non-functional requirements to be guaranteed by the cloud provider. IaaS and PaaS cloud systems still consist of multiple software components that may require updating. In some cases those updating actions are originated by the requirements set in the SLAs of the deployed applications; i.e., the adaptiveness of those IaaS and PaaS systems demands that part of its software were replaced/updated depending on the concrete SLA to be enforced. In this context, further work is still needed in order to:

- Standardize SLA specifications.
- Take a concrete SLA as a basis to configure the set of machines to be used for deploying applications onto IaaS and PaaS systems.
- Adapt the cloud-provider software at run-time, as dictated by the SLA, using dynamic software updating mechanisms.
- Design and implement dynamic software updating mechanisms that automatize the updating process for distributed applications. This can be achieved when the interfaces, states and communication protocols used by the involved components in both software versions are still compatible or there exist clear rules to transform the state being managed in the old version to the new one.
- Include the latter mechanisms in IaaS environments in order to help the application developer in the software update tasks.

5 Conclusion

Software needs to be updated in order to fix remaining bugs, improve its performance or extend its functionality. A static software update procedure consists in a sequence of three steps: (a) stop the software execution, (b) replace the current version of the program with a new one, (c) restart the application. This procedure is valid for single-user applications that are seldom used, since in the long intervals when such applications are not used, only step (b) is needed.

Distributed services that are available to a large set of users require a dynamic software update procedure, since they should be always available. In a dynamic software update, the application requires a non-stopping procedure. Besides this continuity goal, a good updating procedure should be general (i.e., applicable to any kind of software or platform), transparent (users and programmers need not be aware of the software exchange) and robust (the application functionality should be guaranteed during the updating procedure even when problems arise).

Multiple dynamic updating techniques exist nowadays, but most of them are not widely known nor used. This paper has surveyed the existing proposals, analyzing their goals, internal procedures and scope. The availability of modern cloud infrastructures and platforms opens a new horizon to these updating techniques, since both the cloud providers and cloud developers will need dynamic approaches to update and adapt this elastic software. Current dynamic software updating mechanisms facilitate a first solution to this problem. In this scope, the software to be updated is replicated in order to ensure its continuous availability. The update of replicated software requires the combined usage of multiple mechanisms: (a) state transfer or state transformation (in order to translate the managed application state to a format usable by the new software version), (b) version coexistence (for allowing that client requests started with the old version were completed by that old version without being aborted, while other client requests started later were concurrently served with the new version), and (c) scheduling (for deciding a valid replica updating sequence and setting some synchronization points in the interim). However, update automatization and easier adaptation to changes in the SLA are still open problems that will require further improvements in this area.

References

[1] Sameer Ajmani. *Automatic Software Upgrades for Distributed Systems*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 2004.

[2] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular software upgrades for distributed systems. In *European Conf. on Object-Oriented Progr. (ECOOP)*, July 2006.

[3] Joao Paulo Almeida, Marteen Wegdam, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent dynamic reconfiguration for CORBA. In *3rd Intnl. Symp. on Distrib. Objects and Appl. (DOA)*, pages 197–207, 2001.

[4] The Apache Software Foundation. Apache Commons BCEL 6.0, October 2011. http://commons.apache.org/bcel/.

[5] The Apache Software Foundation. Apache Felix, July 2012. http://felix.apache.org.

[6] Filippo Bannò, Daniele Marletta, Giuseppe Pappalardo, and Emiliano Tramontana. Handling consistent dynamic updates on distributed systems. In *IEEE Symp. on Comput. and Commun. (ISCC)*, pages 471–476, June 2010.

[7] Mario R. Barbacci, Dennis L. Doubleday, Charles B. Weinstock, Michael J. Gardner, and Randall W. Lichota. Building fault tolerant distributed applications with Durra. In *Intnl. Wshop. on Config. Distrib. Syst.*, pages 128–139, March 1992.

[8] Preeti Bhoj, Sharad Singhal, and Sailesh Chutani. SLA management in federated environments. In *Intnl. Symp. on Integrated Network Management (IM)*, pages 293–308, Boston, USA, May 1999. IEEE-CS Press.

[9] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos Zarras. A dynamic reconfiguration service for CORBA. In *4th Intnl. Conf. on Config. Distrib. Syst.*, pages 35–42, May 1998.

[10] Toby Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, Massachusetts Institute of Technology, Cambridge, MA, USA, 1983.

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

[12] Aaron B. Brown and David A. Patterson. Rewind, repair, replay: Three R's to dependability. In *10th ACM SIGOPS European Wshop.*, pages 70–77, Saint-Emilion, France, 2002. ACM.

[13] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: A code manipulation tool to implement adaptable systems. France Télécom R+D, DTL/ASR, November 2002.

[14] Bryan Buck and Jeffrey K. Hollingsworth. An API for runtime code patching. *Intnl. J. of High Perf. Comput. Appl.*, 14(4):317–329, 2000.

[15] Navin Budhiraja, Keith Marzullo, Fred B. Schneider, and Sam Toueg. Optimal primary-backup protocols. In *6th Intnl. Wshop. Distrib. Alg. (WDAG)*, volume 647 of *Lect. Notes Comput. Sc.*, pages 362–378. Springer, Haifa, Israel, November 1992.

[16] CGLib Project. CGLib 2.2.2 - Code Generation Library, April 2011. http://cglib.sourceforge.net/.

[17] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful Live Updating System. In *29th Intnl. Conf. on Software Eng. (ICSE)*, pages 271–281. IEEE-CS Press, May 2007.

[18] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic software updating using a relaxed consistency model. *IEEE Trans. Software Eng.*, 37(5):679–694, September 2011.

[19] Junqing Chen and Linpeng Huang. Dynamic service update based on OSGi. In *WRI World Congress on Software Eng. (WCSE)*, volume 3, pages 493–497, Xiamen, China, May 2009. IEEE-CS Press.

[20] Shigeru Chiba. Javassist 3.16.1, March 2012. http://www.csg.ci.i.u-tokyo.ac.jp/˜chiba/javassist/.

[21] Robert C. Daley and Jack B. Dennis. Virtual memory, processes, and sharing in MULTICS. *Commun. ACM*, 11(5):306–312, 1968.

[22] Xavier Défago, André Schiper, and Péter Urbán. Total order broadcast and multicast algorithms: Taxonomy and survey. *ACM Comput. Surv.*, 36(4):372–421, 2004.

[23] M. Dmitriev. Towards flexible and safe technology for runtime evolution of Java language applications. In *Wshop. on Eng. Complex Object-Oriented Syst. for Evolut.*, Tampa, Florida, USA, November 2001.

[24] The Eclipse Foundation. Equinox, 2012. http://eclipse.org/equinox/.

[25] Elmootazbellah N. Elnozahy, Lorenzo Alvisi, Yi-Min Wang, and David B. Johnson. A survey of rollback-recovery protocols in message-passing systems. *ACM Comput. Surv.*, 34(3):375–408, 2002.

[26] Ericsson AB. Erlang Programming Language, 2012. http://www.erlang.org.

[27] Clément Escoffier, Didier Donsez, and Richard S. Hall. Developing an OSGi-like service platform for .NET. In *IEEE Consum. Commun. and Network. Conf. (CCNC)*, volume 1, pages 213–217, January 2006.

[28] Robert S. Fabry. How to design a system in which modules can be changed on the fly. In *2nd Intnl. Conf. on Software Eng. (ICSE)*, pages 470–476, San Francisco, CA, USA, 1976. IEEE-CS Press.

[29] Ophir Frieder and Mark E. Segal. On dynamically updating a computer program: from concept to prototype. *J. Syst. Software*, 14(2):111–128, February 1991.

[30] Cristiano Giuffrida and Andrew S. Tanenbaum. A taxonomy of live updates. In *Adv. School for Comput. and Imaging Conf. (ASCI)*, Veldhoven, The Netherlands, November 2010.

[31] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Addison-Wesley, 3rd edition, 2005.

[32] Hannes Goullon, Rainer Isle, and Klaus-Peter Löhr. Dynamic restructuring in an experimental operating system. *IEEE Trans. Software Eng.*, 4(4):298–307, 1978.

[33] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic update of Java applications—balancing change flexibility vs programming transparency. *J. Softw. Maint.-Res. Pr.*, 21(2):81–112, March 2009.

[34] Allan Raundahl Gregersen, Michael Rasmussen, and Bo Nørregaard Jørgensen. Javeleon 2.0, June 2012. http://javeleon.org.

[35] Allan Raundahl Gregersen, Douglas Simon, and Bo Nørregaard Jørgensen. Towards a dynamic-update-enabled JVM. In *Wshop. on AOP and Meta-Data for Softw. Evol.*, Genova, Italy, 2009. ACM.

[36] Rachid Guerraoui and André Schiper. Software-based replication for fault tolerance. *IEEE Computer*, 30(4):68–74, 1997.

[37] Deepak Gupta. *On-line Software Version Change*. PhD thesis, Dept. of Comput. Sc. and Eng., Indian Institute of Technology, Kanpur, India, November 1994.

[38] Deepak Gupta and Pankaj Jalote. On line software version change using state transfer between processes. *Software Pract. Exper.*, 23(9):949–964, September 1993.

[39] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A formal framework for on-line software version change. *IEEE T. Software Eng.*, 22(2):120–131, February 1996.

[40] Richard S. Hall and Nektarios K. Papadopoulos. Oscar - OSGi Framework, May 2005. http://forge.ow2.org/projects/oscar/.

[41] Maurice P. Herlihy and Barbara Liskov. A value transmission method for abstract data types. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 4(4):527–551, October 1982.

[42] Michael Hicks. *Dynamic Software Updating*. PhD thesis, University of Pennsylvania, 2001.

[43] Michael Hicks, Jonathan T. Moore, and Scott Nettles. Dynamic software updating. In *ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation*, PLDI '01, pages 13–23, Snowbird, Utah, United States, May 2001. ACM.

[44] Michael Hicks and Scott Nettles. Dynamic software updating. *ACM T. Progr. Lang. Sys.*, 27(6):1049–1096, November 2005.

[45] Christine R. Hofmeister and James M. Purtilo. A framework for dynamic reconfiguration of distributed programs. Technical Report UMIACS-TR-93-78, University of Maryland, College Park, 1993.

[46] Christine Ruth Hofmeister. *Dynamic Reconfiguration of Distributed Applications*. PhD thesis, University of Maryland at College Park, College Park, MD, USA, 1993.

[47] ISO. International standard ISO/IEC 10746-1:1998(E): Information technology - open distributed processing - reference model: Overview. International Standard Organization, Case Postale 56, CH-1211 Genève 20, Suiza, December 1998.

[48] Java-Source.net. Open source bytecode libraries in Java, July 2012. http://java-source.net/open-source/bytecode-libraries.

[49] The Knopflerfish Project. Knopflerfish OSGi - Open source OSGi service platform, July 2012. http://www.knopflerfish.org/.

[50] Jeff Kramer and Jeff Magee. The evolving philosophers problem: Dynamic change management. *IEEE T. Software Eng.*, 16(11):1293–1306, Nov. 1990.

[51] Eugene Kuleshov. Using the ASM framework to implement common Java bytecode transformation patterns. In *6th Intnl. Conf. Aspect-Oriented Softw. Devel. (AOSD)*, Vancouver, Canada, March 2007.

[52] Insup Lee. *DYMOS: A Dynamic Modification System*. PhD thesis, Dept. of Comput. Sc., Univ. of Wisconsin, Madison, 1983.

[53] Tim Lindholm and Frank Yellin. The Java virtual machine specification, second edition, 1999.

[54] Scott Malabarba, Raju Pandey, Jeff Gragg, Earl T. Barr, and J. Fritz Barnes. Runtime support for type-safe dynamic Java classes. In *14th European Conf. on Object-Oriented Progr. (ECOOP)*, volume 1850 of *Lect. Notes Comput. Sc.*, pages 337–361, Cannes, France, June 2000. Springer.

[55] Emili Miedes and Francesc D. Muñoz-Escoí. Dynamic switching of total-order broadcast protocols. In *Intnl. Conf. Paral. Distrib. Process. Tech. Appl. (PDPTA)*, pages 457–463, Las Vegas, Nevada, USA, July 2010. CSREA Press.

[56] Marco Milazzo, Giuseppe Pappalardo, Emiliano Tramontana, and Giuseppe Ursino. Handling runtime updates in distributed applications. In *ACM Symp. on Applied Comput. (SAC)*, pages 1375–1380, Santa Fe, New Mexico, 2005. ACM.

[57] David Mosberger. Memory consistency models. *Operating Systems Review*, 27(1):18–26, 1993.

[58] Yogesh Murarka and Umesh Bellur. Correctness of request executions in online updates of concurrent object oriented programs. In *15th Asia-Pacific Software Eng. Conf. (APSEC)*, pages 93–100. IEEE-CS Press, December 2008.

[59] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical dynamic software updating for C. In *ACM SIGPLAN Conf. on Progr. Lang. Design and Impl. (PLDI)*, pages 72–83, Ottawa, Ontario, Canada, 2006. ACM.

[60] ObjectWeb. ASM 4.0, October 2011. http://asm.ow2.org/.

[61] OSGi Alliance. About the OSGi service platform. Technical whitepaper. Revision 4.1, June 2007.

[62] OSGi Alliance. OSGi Alliance Home Page, July 2012. http://www.osgi.org.

[63] Valerio Panzica La Manna. Dynamic software update for component-based distributed systems. In *16th Intnl. Wshop. on Component-Oriented Progr. (WCOP)*, pages 1–8, New York, NY, USA, 2011. ACM.

[64] James M. Purtilo. The POLYLITH software bus. *ACM T. Progr. Lang. Sys.*, 16(1):151–174, January 1994.

[65] James M. Purtilo and Christine R. Hofmeister. Dynamic reconfiguration of distributed programs. In *11th Intnl. Conf. on Distrib. Comput. Sys. (ICDCS)*, pages 560–571, May 1991.

[66] Jan Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed applications through software modularization. In *Middleware*, volume 4834 of *Lect. Notes Comput. Sc.*, pages 1–20, Newport Beach, CA, USA, 2007. Springer.

[67] Jan S. Rellermeyer and Gustavo Alonso. Concierge: A service platform for resource-constrained devices. In *2nd ACM European Conf. on Comput. Syst. (EuroSys)*, pages 245–258, Lisbon, Portugal, March 2007. ACM.

[68] Tobias Ritzau and Jesper Andersson. Dynamic deployment of Java applications. In *Java for Embedded Systems Workshop*, London, United Kingdom, May 2000.

[69] María Idoia Ruiz-Fuertes and Francesc D. Muñoz-Escoí. Performance evaluation of a metaprotocol for database replication adaptability. In *28th Intnl. Symp. Reliab. Distrib. Syst. (SRDS)*, pages 32–38, Niagara Falls, New York, USA, September 2009. IEEE-CS Press.

[70] Holger Schmidt, Jan-Patrick Elsholz, Vladimir Nikolov, Franz J. Hauck, and Rüdiger Kapitza. OSGi4C: Enabling OSGi for the cloud. In *4th Intnl. Conf. on Commun. Syst. Software and Middleware (COMSWARE)*, Dublin, Ireland, June 2009. ACM.

[71] Fred B. Schneider. Implementing fault-tolerant services using the state machine approach: A tutorial. *ACM Comput. Surv.*, 22(4):299–319, 1990.

[72] Mark E. Segal and Ophir Frieder. Dynamic program updating in a distributed computer system. In *Conf. of Software Maintenance*, pages 198–203, Scottsdale, AZ, USA, October 1988.

[73] Mark E. Segal and Ophir Frieder. On-the-fly program modification: Systems for a dynamic updating. *IEEE Software*, 10(2):53–65, 1993.

[74] Marcin Solarski. *Dynamic Upgrade of Distributed Software components*. PhD thesis, Fakultät IV (Elektrotechnik und Informatik), Technische Universität Berlin, 2004.

[75] Marcin Solarski and Hein Meling. Towards upgrading actively replicated servers on-the-fly. In *26th Intnl. Comput. Software and Appl. Conf. (COMPSAC)*, pages 1038–1043, 2002.

[76] Nigamanth Sridhar, Scott M. Pike, and Bruce W. Weide. Dynamic module replacement in distributed protocols. In *23rd Intnl. Conf. on Distrib. Comput. Sys. (ICDCS)*, pages 620–627, May 2003.

[77] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and predictable dynamic software updating. *ACM T. Progr. Lang. Sys.*, 29(4), August 2007.

[78] Swiss Federal Institute of Technology Zurich. Concierge OSGi - An optimized OSGi R3 implementation for mobile and embedded systems, April 2009. http://concierge.sourceforge.net/.

[79] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A bytecode translator for distributed execution of "legacy" Java software. In *European Conf. on Object-Oriented Progr. (ECOOP)*, pages 236–255. Springer, 2001.

[80] Andre L. C. Tavares and Marco Tulio Valente. A gentle introduction to OSGi. *SIGSOFT Software Eng. Notes*, 33(5), September 2008.

[81] L. A. Tewksbury, Louise E. Moser, and P. Michael Melliar-Smith. Live upgrades of CORBA applications using object replication. In *IEEE Intnl. Conf. on Software Maintenance (ICSM)*, pages 488–497. IEEE-CS Press, 2001.

[82] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates. *IEEE T. Software Eng.*, 33(12):856–868, December 2007.

[83] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A break in the clouds: Towards a cloud definition. *SIGCOMM Comput. Commun. Review*, 39(1):50–55, January 2009.

[84] Ximei Wang, Shoubao Yang, Shuling Wang, Xianlong Niu, and Jing Xu. An application-based adaptive replica consistency for cloud storage. In *9th Intnl. Conf. on Grid and Coop. Comput.*, pages 13–17, Nanjing, November 2010.

[85] ZeroTurnaround. JRebel 4.5.4, January 2012. http://zeroturnaround.com/jrebel/.