Dynamic Software Update

Emili Miedes and Francesc D. Muñoz-Escoí

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

{emiedes, fmunyoz}@iti.upv.es

Technical Report ITI-SIDI-2012/004

Dynamic Software Update

Emili Miedes and 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-2012/004

e-mail: {emiedes, fmunyoz}@iti.upv.es

May 21, 2012

1 Introduction

Software systems are continuously evolving. Some typical examples of software changes may be changing the implementation of a given service, adding a new service, removing an existing one or fixing a bug or a security vulnerability.

The classic way to apply a change to a software system that is currently running consists in producing a new version of the software, stopping the installed version of the software, removing it, installing the new version and restarting it.

This procedure has a number of drawbacks. First, it forces the *unavailability* of the service offered by the software. Moreover, from the client side, it forces the *restart* of the client software that was accessing the software. Furthermore, it complicates the design and development of the software service. For instance, the software must be able to handle update requests, probably save some state to a persistent device and switch itself off. When the next version is started up, it must retrieve the persisted stated, use it to initialize itself and finally go on providing its service.

The alternative is the use of a *dynamic update mechanism* which allows a software system to be updated *dynamically*, this is, without requiring it to be switched off and on again, thus avoiding the issues pointed out above.

Nowadays, such mechanisms are useful for many types of software systems and applications. First, they are useful for common final user desktop applications to transparently apply regular updates and bug fixes, without forcing the user to restart the application.

Second, they are useful for updating and upgrading the operating systems themselves, this is, 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 more wide scale context, dynamic software update mechanisms are useful to update any type of web service or application that offers a 24/7 service to a potentially large set of users. Without a dynamic update mechanism, to update such an application, a stop-and-restart model would be used, which causes significant nuisances to the user and may cause a significant harm to the holders of the application. First, the ongoing user requests must be *aborted*, thus causing a significant nuisance to the connected users, which sooner or later turns out to have a negative impact on the entity responsible of the service. Moreover, the application must be kept inactive during the time needed to perform the update or upgrade and the corresponding testing, this yielding it unavailable so it can not serve new user requests, which definitely 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 application deployed

in the cloud. Nevertheless, all the cloud providers run a software infrastructure 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 a number of authors, in different contexts, and a number of techniques and solutions of different types have been proposed. During that time, few surveys of dynamic update mechanisms have been published, too. 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 order some of the concepts and techniques found in the literature of dynamic software updating. First, in Section 2 we propose a selection of requirements and goals we identify as being *basic* in any dynamic software update mechanism. Then, in Section 3 we identify a number of techniques used in the existing literature. In Section 4 we discuss a number of issues related to some of the topics covered in the preceeding 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 dynamically. This means that to apply a change in the software, it is no longer necessary to stop the system and restart it once updated. Instead, the update is applied in run-time.

In the existing literature related to dynamic software updating, we found a variety of authors that provide their own *definition* of dynamic software update and list the requirements and goals that a dynamic update mechanism may have. In this section we identify a number of such requirements and goals. For each requirement, we describe the main issues and provide some literature references in which the topic is somehow covered. In some cases, the authors propose slight variations.

Continuity and Minimal Disruption. *The update can be performed in run-time, without stopping and restarting the system to update and it does not interrupt the execution of the software for a too long period of time*.

The first part of the requirement (the avoidance of a stop and restart) is the *essential* concept in the *dynamic software update* topic, as explained in Section 1 and all the references that cover the dynamic update software just implicitly assume it. Some of the authors that identify it explicitly are Fabry [29], Segal and Frieder [63, 32], Solarski [64], Murarka and Bellur [54] and Gregersen and Jørgersen [35].

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 or 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, we can consider two *worst cases* that may happen in a *continuous* update mechanism (this is, one that avoids stopping and restarting the software). 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 completely unavailable. In the second case, the update process is also dynamically applied and can be executed in parallel with the service but it reduces the performance of the latter 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 performance of the service.

Such a best case is quite difficult to achieve so many authors consider a *relaxed version*. In some cases, 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 ([48, 63, 32, 43, 24, 35, 52]). In other cases, this requirement is more specific, like in [29], which admits a *momentary delay* in the normal execution of user requests or [64] which accepts that the update process may interrupt the application *the shortest time possible*.

Moreover, some authors require the system to upgrade to be in a *quiescent* state for the update to be performed, while others allow to apply a dynamic update while the software is fully operative. In Section 3.1 we review some issues related to the concept of *quiescence*.

Transparency. *The update process is transparent*, which means that it has no significant impact on its context (the user, the programmer and the managed application) beyond the results it provides (a dynamic update). Again, this is a case of a manifold requirement, since several types of transparency can be considered.

First, we can consider the *user transparency*, the transparency from the point of view of the final user. According to it, in an ideal case, the update mechanism is *hidden* to the user, this is, the user does not need to be aware of the update mechanism. Moreover, it does not require the user to interact with the application in any specific manner or have any specific knowledge or skills. In the worst case, the user needs to know about the update mechanism and it changes the way the user interacts with the software.

Then, we can consider the transparency regarding the programmer's point of view. A *programmer transparent* update procedure is one that does not require the programmers to have specific knowledge about the update process itself and does not change the way they design and develop the systems.

Moreover, the update process can also be *application transparent*, this is, transparent from the point of view of the software itself. Ideally, the update mechanism is one that does not impose any constraint to the program about how to be designed or implemented, does not change the expected behavior of the program, does not impose any noticeable performance impact or any other constraint and is not noticeable to those parts of the system that are not related to it. Typical examples of constraints that may be required to the updateable software components of an application are the use of specific programming or configuration languages, interfaces or base classes and libraries to include in the application.

Regarding the literature, these transparency requirements are identified by several authors. The *user transparency* requirement, as expressed above, is not found in any of the references surveyed although we can consider that all those references that admit a *small* disruption in the correct operation of the update mechanism are implicitly using a *relaxed* form of user transparency. On the other hand, Gregersen and Jørgersen [35] explicitly require *programmer transparency* and Solarski [64] and Bannò *et al.* [16] 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 types 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 removing some existing components. In the general case, 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.).

The first interpretation is the one used by Ajmani [11, 12] and Gregersen and Jørgersen [35] while the second interpretation is used by Solarski [64]. Moreover, Panzica [52] provides a classification of dynamic updates.

Consistency and Integrity. *The update of a component leaves it and the whole application in a consistent or correct state.*

This requirement also has some variants. Generally speaking, the main variant is related to the state of the software after a dynamic update is applied and requires that once the update has been applied, the software is in a state *similar* to the one that would be got if the update had been applied statically. Moreover, after the dynamic update, the software is equally able to go on serving user requests.

A second variant of the requirement is related to the proper termination of the pending user requests. Ideally, the requests that are interrupted by a dynamic update are properly terminated and the state of the

software is likewise correct.

In the literature, some authors identify this requirement in a *vague* manner. For instance, Kramer and Magee [48], Sridhar *et al.* [66], Solarski [64], Murarka and Bellur [54] require that the update process leave the system in a *consistent* or *correct* state but do not elaborate too much about the concept of *consistency* or *correctness*.

Gregersen and Jørgersen [35] are a bit more specific and require the state of the software 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.* [16] require *data consistency* and also *consistency of flow* (the proper termination of pending requests). Finally, Panzica [52] identifies both variants of *consistency* pointed out above.

State Preservation. *The update of a component preserves as much of its state as possible.*

When a *classic stop-and-restart* deployment model is used, the software system is stopped, which means that its state is lost unless it is previously saved to some persistence device. Sometimes, the user is made responsible of doing the task. Nevertheless, the use of a *dynamic update* mechanism does not directly guarantee that the state kept by the old version of the application is preserved so a specific requirement to keep the state of the application is needed.

Thus, the update mechanism must provide some way to capture the state of the component to update and *preserve it* in some way, to ensure that when a dynamic update is applied to an application, the state it had just before the update is *transferred* to the new version so it can operate with it. The state transfer may include the transformation steps required to *adapt* the data formats understood by the previous version of the application to the formats used by the new version.

In Section 3.5 we review some issues related to state transfer and transformation functions, respectively.

Some authors declare this requirement explicitly, like Sridhar *et al.* [66] and Ajmani *et al.* [11, 12] who also consider the possibility of applying the necessary transformations to the data.

Version Coexistence. *The update process allows a component that has been updated to coexist with an old version of the same component.*

This requirement is important from a practical point of view. For instance, in a client/server application in which the server is dynamically updated, this requirement helps to ensure a *smooth* transition of the set of clients that send requests to the server. If the server only keeps one version of the updateable components, when they are updated the whole set of clients may be *forced* to restart, in order to be able to communicate with the new versions of those components (unless some indirection level is used among the clients and the server side, as pointed out in Section 3.3). The coexistence of old and new versions of the updateable components running in the server allows to keep the clients alive and let them go on working normally until they are shut down. New clients started after the update would access the new version of the updated components.

Moreover, the coexistence of versions also allows the clients to be dynamically updated (so there is no need to shut them down). The old-version clients can issue their requests to the old-version components of the application. Once a client is dynamically updated, it can issue its requests to the corresponding new versions of the components.

On the other hand, this requirement is also important from the point of view of the server side, especially in distributed applications and, in particular, in replicated systems. In such a case, besides the *intra-node* version coexistence requirement explained above, we can also consider an *inter-node* version of the requirement, by which *the update process allows that the new versions of a component that is replicated in a number of nodes of a distributed system coexist with old versions of the same component running in some other nodes of the system.*

This requirement allows to perform the update of the distributed nodes in stages, for instance, updating a few at a time, instead of being forced to update them all at once. In small-scale distributed systems, this is just a useful feature but it turns to be essential in medium to large-scale systems, in which it is not possible to update all the nodes at once.

Few authors, like Ajmani *et al.* [11, 12] (they call it *mixed mode operation*) and Solarski [64] include a requirement similar to this one.

Other Requirements. We can cite some other requirements identified by some authors in the related literature.

Atomicity and *Rollbackness*. *The update mechanism isolates the execution of the dynamic updates, respect the execution of user requests and other dynamic updates. Moreover, the update mechanism is able to rollback a given update.* The rollback ability is important because it allows to rollback a given update when it is found faulty or must be uninstalled for any other reason.

In Solarski [64], 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.* [24, 25], besides updating a component to the next version it is also possible to apply an *update* to go back to a previous version, which actually is an effective rollback mechanism. Gregersen and Jørgersen [35] also consider the ability to *rollback* an update to restore the previous version of the software. Finally, Bannò *et al.* [16] also consider the update of a number of components as an atomic action although they never mention the ability to rollback an update.

Schedulability and Automation. *The update mechanism allows to schedule the updates or provides some other ways to automate them.* Some authors (like Ajmani *et al.* [11, 12]) consider the possibility to schedule the execution of the updates to apply or more generally, provide some automation mechanisms to minimize the human intervention (for instance, Segal and Frieder [63, 32], Solarski [64] and Panzica [52])

Simultaneous Updates. *The update mechanism allows to apply more than one update simultaneously.* This is considered by Segal and Frieder [63, 32] and Solarski [64].

3 Concepts and Techniques

In this section we identify a number of concepts and techniques used and found in the surveyed references and somehow related with dynamic software updating.

3.1 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, can not 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). For this, different authors try to ensure that the component to update reaches some *stable* state. Depending on the author, this stability requirement is given a different name and described in different ways and a number of mechanisms can be used to enforce it.

Search in the Execution Stack. Some authors inspect the execution stack of a process in order to know if a given function (or procedure) of a program 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 the function (by redirecting the calls as in Section 3.3, applying a binary patch as in Section 3.2, etc.).

This technique is usually part of some other technique or update procedure. For instance, Gupta *et al.* [40] inspect the stack to know if a given function or procedure can be updated. They also use it in [39] to perform its state transfer procedure (see Section 3.5 for additional information on state transfer).

Segal and Frieder [63, 32] also inspect the stack of the process to update 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 [58, 45, 57]. They propose the use of an abstract format to represent the *frames* of the execution stacks of regular processes and implement 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. This can be achieved by making the program to enter a given *idle* function or procedure. 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. For instance, in Gupta *et al.* [40], 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 [40]). 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 [26] use the same idea in the context of dynamic update of OSGi applications (also see Section 3.6.3). 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 [33] also use a similar approach in their proposal of an operating system-oriented dynamic update procedure. It uses 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 [48] 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 of the system has to be updated, it is forced to *passivate*, this is, to reach a passive 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 the *passive set* of the given node (this is, all the 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 [48] 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 the components in the system would have to be passivated, which may lead the application to an unavailability state, which is totally contrary to the essence of any dynamic software update mechanism.

For instance, Vandewoude *et al.* [71] argue that the *quiescence* concept in [48] 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 update process.

To understand the differences between quiescence and tranquility, one must compare the formal definition of the *quiescent* and *tranquil* states, according to [48] and [71], 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 [71], the c) clause of the *quiescence state* definition implies that a node may be either *actively processing a request* or *waiting*

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

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 [71], 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 3.3 for other examples that use some kind of intermediary level). For instance, some sort of *central* update *manager* or *intermediary* proxies may be used to 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.* [16] in their FREJA framework. They use several types of intermediary Java objects. On the 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: [20, 17, 44, 19, 61, 14, 70, 42]. 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 [27] 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). This mechanism is not mature enough to be considered production-ready yet. 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*.

3.2 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 being 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 to be used by a dynamic update mechanism. Nevertheless, it has a number of disadvantages.

This technique is strongly dependent on the particular compiler and especially on the hardware architecture it is aimed to. It also requires from the designers and programmers a deep knowledge in low level details like the exact machine language used by the target processor. To apply an update to a program, to update its version v to version $v+1$, the programmer must know the exact binary representation of both the

code to replace and the new code. Below we cite some alternatives that avoid this last restriction although they still require some deep low level knowledge to be applied.

This technique has some other disadvantages, derived from its low level nature. For instance, 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.

One of the first authors to propose the use of *binary redirection* was Fabry [29]. 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 3.3) 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.

Hicks and Nettles [43] 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 3.3), 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 client 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 the next version to apply, some automated tool is used to compute 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.

The authors have a prototype implementation of the proposed framework. They have also implemented an *updateable web server* (FlashEd) and used it to test the operation of the dynamic update framework implementation.

Chen *et al.* [24, 25] describe POLUS, a tool that offers support to dynamically update a software system. Roughly speaking, 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 (see Section 3.6.1 for other proposals that use some sort of static analysis of the source 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 3.3 for other forms of level indirection).

The use of dynamic patches is inspired by Hicks and Nettles [43], although POLUS is distinguished by the possibility to *reverse* this procedure (also see Section 3.6.9). 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 3.6.9 other examples of rollback-enabled mechanisms are given.

Binary Rewriting in Java. Another particular case of binary rewriting is its application to Java programs. From an abstract point of view, 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 ([34, 50]). The modifications are typically expected to preserve the *Java binary compatibility* ([34]).

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. For instance, Milazzo *et al.* [53] study the run-time update of distributed applications written in the Java programming language. They 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 3.3). This layer includes a new Java classloader that uses some Java rewriting techniques to modify the Java bytecode in 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 [35] 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 [35] 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 3.3 for other techniques based on adding some *indirection* level.

They also need to manipulate the Java bytecode, in a number of ways to, generally speaking, prepare both client and server code to use and be used by the update mechanism. Other authors also use some binary-level rewriting techniques.

The update procedure also includes a *lazy state migration* that is used to transfer 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 *field* is transferred individually, when it is accessed by the first time.

Their proposal also allows the rollback of applied updates (see Section 3.6.9 for other authors that also offer some sort of *rollback* mechanism).

Bannò *et al.* [16] 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 [16]). 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 3.3).

On the other hand, there are currently available a number of tools and libraries that offer services related to bytecode manipulation (including run-time manipulations). For the Java programming language, there are many alternatives like ObjectWeb ASM [56, 23, 49], CGLIB [7], Javassist [9, 68], Apache Commons BCEL [31], Javeleon [8, 36], JRebel [74] and some others listed in [4].

3.3 Use of Proxies, Intermediaries and Indirection Levels

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. These techniques are useful in client/server systems in which there are a number of dynamically updateable servers offering some service and also a number of clients that issue requests to the former.

The basic idea consists in adding an intermediary level between a client and the dynamically updateable server it is accessing. Instead of having the client directly call the functions and procedures that implement the service, it calls some intermediary code that points to the current implementation of the service. Such an intermediary code can be dynamically overwritten (see Section 3.2).

This approach has been used by a number of authors. For instance, Fabry [29] was one of the first to use it, in combination with two different binary-level overwriting.

Bloom's Ph. D. thesis [20], reuses the idea of redirecting the calls to the updateable code by *remapping* some handlers, in the context of Argus programs.

Segal and Frieder [63, 32] use *interprocedures*, which are some intermediary procedures used to *redirect* the client invocations to *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. The authors argue that this approach is feasible under those hardware architectures that offer an *indirect addressing mode* like those provided by the Motorola MC68020 processor or the Intel's 386 architecture.

In POLUS [24, 25], 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 the intermediary level, proxy objects that *simulate* the real implementation of the service. The idea is that the client code does not call the objects that implement the service but uses intermediary proxies. These offer to the client code the same interface than the original service objects and hide the complexity of the dynamic update. There are a number of authors that follow this approach.

For instance, Purtilo *et al.* [58, 57] 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 [66] 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#).

In [53], Milazzo *et al.* propose a mechanism to dynamically update regular Java applications by means of using an intermediary layer between the Java service classes and some client code that issues invocations to the former. This layer includes some new interfaces and classes created and instantiated in loading time.

Ajmani *et al.* [12] use some intermediary objects called *simulation objects* used to 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.

Gregersen and Jørgensen [35] use some intermediary proxies, that are dynamically generated, to manage the process of class loading and intercept and redirect the invocations to the *service* objects.

Chen and Huang [26] propose the use of intermediary dynamic proxies in the context of dynamic update of OSGi applications. These proxies would be placed among the updateable service bundles and the client code, this hiding to the later the existence of dynamic updates.

In their framework FREJA, Bannò *et al.* [16] also use some specific Java class loaders and some intermediary objects to control the execution of updateable components.

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. The goal is to allow a managed application to cooperate with the update mechanism. This *intrusion* can take different forms.

A first type of *intrusion* consists in defining special functions or procedures in both the update mechanism and the application to update. The idea is that, on the one hand, the application to manage 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 [48] 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 [33], 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 3.1). 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 or 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 3.6.3)

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 [32] and by Neamtiu *et al.* [55], 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 [43] who propose a mechanism that allows the programmer to *mark* places in the code that should not be interrupted by a dynamic update.

There are other examples in which meta-information is provided to achieve some other goals. For instance, Bannò *et al.* [16] identify the need to design the updateable applications in a special way and provide some meta-information to the update mechanism for this to be able to preserve the *semantic consistency* of the application to update.

Giuffrida and Tanenbaum [33] also argue that the best approach to build dynamically updateable systems consists in making them aware of the dynamic update process and require that the application provides some meta-information that may be used by the update mechanism to lead the application to a quiescent state (also see Section 3.1), before applying a dynamic update.

To conclude, we can say that, in principle, the use of *intrusion mechanisms* offers both the update mechanisms and the managed applications the possibility to cooperate in the application of the dynamic updates. Nevertheless, it must be considered that such *intrusion mechanisms* reduce the level of *transparency* offered by the update mechanism, especially the *application transparency* (see the *transparency*

requirement in Section 2). Indeed, forcing the application to provide some specific functions, adapt to a specific architecture, have some special marks, etc. makes it dependent on the update mechanisms and also makes the latter less transparent to the application.

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, etc.) and the next version, in order not to lose it when the update is applied.

Some of them 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 may be called and some *serializable* representation of the state may be got. 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.

In his Ph.D. thesis, Bloom [20] 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.* [58, 57] propose the use of an abstract representation of the data kept by the (dynamically reconfigurable) modules of the systems 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 have used state transfer techniques in their update mechanisms [39, 65, 64, 66, 35, 26].

Bannò *et al.* [16] identify the need of the *consistency of the data* in a dynamic update and the transfer of the data from the current component to the updated one.

On the other hand, 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* to transform the state of a component in the format used by a given version to the proper format. These functions are typically provided by the programmer, like in [20, 32, 58, 43, 12, 67, 54, 26].

3.6 Other Issues

In this section we briefly review some other issues related to dynamic software update.

3.6.1 Source Code Static Analysis

In a number of papers, some kind of *static analysis* of the application source code is performed, according to different objectives. Some authors use it to know in which points of the programs is safe to perform a dynamic update or in which ones an update should not be performed at all. The key idea is to *protect* the state of the component or program so the update does not yield the component or program in an inconsistent state. For instance, it is safe to apply an update during the execution of a *read-only* function or procedure (this is, one that does not alter the state of the program). It is also safe to apply it in the very beginning of the execution of a regular function, before it modifies any part of the program's state. On the other hand, it may not be safe to apply an update during the execution of a regular function since it may be changing the state of the program. Such an *interrupting* update hinders and can even avoid a proper state transfer and reconstruction.

Other authors compare the source code of the current version of a program with the next version and build a *patch* out of the differences, to be applied dynamically. In some cases, the analysis can be completely automated while in others it is a manual or human-assisted process.

For instance, Stoyle *et al.* [67] and Murarka *et al.* [54] propose the static analysis of the source code to identify points in which a dynamic update may or may not be applied, while ensuring some *correctness* property.

In their Proteus system, Stoyle *et al.* [67] propose a property called *conn-t-freeness* and tries to ensure that *after a dynamic update of some type t, no updated value v' of type t will ever be manipulated by code that relies on the old representation of t*. Their *static updateability analysis* is used to label points in the program with *update expressions* that identify those types t for which the program may not be *conn-t-free*. This information is used by the Proteus run-time system to know if a given type can be dynamically updated in a given point of the program.

Murarka *et al.* [54] also perform an analysis of the source code of both the current version of the program and its next version, to ensure that their *Request Execution Criteria* is fulfilled. At least one of the following conditions must be satisfied. The *New Program Execution* condition requires that, before an update to a given class, no request accesses an object of the old version of the class that could not be accessed with the new definition of the class. Moreover, the *Old Program Execution* condition requires that, after an update to a given class, no request accesses an object of the old version of the class that can not be accessed with the new definition of the class and no request accesses an object of the new version of the class that can not be accessed with the old definition of the class. Thus, the result of the analysis is a set of *update points* and *unsafe regions* in which dynamic updates may or may not be applied.

Neamtiu *et al.* propose Ginseng [55], a dynamic update solution for programs written in C, based in dynamic patching (see Section 3.2). In their solution, they depend on some static analysis of the source code to ensure that the updates are *type-safe*. The idea is to ensure a *representation consistency* property, by which, at any moment, any value of a type T is a member of the last version of T (which means that two different versions of the same type will never coexist).

Roughly speaking, the procedure is the following. The programmer is expected to identify *update points* in the program, in which a dynamic update may be applied. When the Ginseng compiler builds a dynamic patch to apply the next version of the program, it annotates each of those update points with information about *which types should not be updated* by a dynamic update applied in each update point. Later, when a patch is applied in run-time, the annotations are *checked* to ensure that the update does not violate the *representation consistency* property.

Altekar *et al.* propose OPUS [15], which also depends on a similar analysis to detect *unsafe dynamic updates* and other authors, like Hicks and Nettles [43] and Chen *et al.* [24, 25] in their POLUS system also use the source code of the old and new versions of a component to update to build a *patch* that will be applied dynamically.

On the other hand, static analysis has also been used for other purposes. For instance, Bauml and Brada [18] propose a procedure based on the static analysis of source code to automatically decide the `a.b.c`-like version string of the next version of an application version. As usually, incompatible changes force a change of the major version, while backwards compatible changes only cause the change of the minor version number. The micro version number only changes when internal implementation-related changes are made.

More specifically, the analysis identifies changes between two given versions v_1 and v_2 of any type, according the hierarchy relationship between them. First, they could be the same type. Also the change is a *Specialization* or a *Generalization* if v_2 is a *specialization* or a *generalization* of v_1, respectively. Finally, the change is a *Mutation* if there is no subtype relation between v_1 and v_2.

Considering all the types of changes identified by the analysis, some rules are applied to globally classify the update of the program. If there is at least one Mutation, or there is some *Specialization* and some *Generalization*, then the global *level* of the update is a *Mutation*, and the major version number is incremented. Otherwise, if there is some *Generalization*, the change is a *Generalization* and the major version is incremented as well. Otherwise, if there is some *Specialization* the change is a *Specialization* and the minor version is incremented. Otherwise, it is considered that no significant change is done and only the micro version number is incremented.

3.6.2 Use of Underlying Facilities

A number of authors base their proposals on features of a given underlying *infrastructure*: a given hardware architecture, a programming methodology or paradigm, an ad-hoc programming or configuration language, a specific general-purpose programming language or any other specific base level.

For instance, at a low abstraction level, the solution proposed by Frieder and Segal [63, 32] needs that the hardware architecture of the underlying machine offers an *indirect addressing mode*. Gupta and Jalote's proposal [39] was also designed to work on a specific hardware and software platform (SunOS running on a Sun 3/60 workstation) and also depends on a specific feature of the hardware architecture (specifically, the *segment-based memory addressing* mode). In practice, the requirement of such features is not a significant constraint since these features are available in common processors (as they were at the time of [63, 32, 39]).

Other authors propose solutions that are a bit more general and can be used with programs written in *imperative languages*, like the one by Hicks and Nettles [43].

There are also some authors who develop their proposal based on their own infrastructure. For instance, Kramer and Magee base their proposal [47, 48] on their CONIC configuration language and infrastructure [46]. In Proteus, Stoyle *et al.* [67] describe a dynamic software update solution based on its own programming language, compiler and run-time, among other tools and resources. In his Ph. D. thesis, Bloom [20] proposes a dynamic update solution for programs written with the Argus programming language ([51]). For the C programming languages there are some options, like the proposal in Gupta's Ph. D. thesis [38], Ginseng by Neamtiu *et al.* [55] and POLUS by Chen *et al.* [24, 25].

In a higher level of abstraction, regarding the Java programming language and Virtual Machine (JVM), there are a large number of references. First, some authors propose dynamic update solutions for Java programs (for instance, [61], [53], [35], and [16]). In Section 3.2 a number of technologies related to dynamic software update in Java are cited. Moreover, Dmitriev [27] studies an existing mechanism available in the HotSpot JVM to allow the dynamic update of Java code in debugging time. Some other authors like Gregersen and Jørgensen [37] propose dynamic update mechanisms based on modifying the standard JVM. Nevertheless such an approach presents a number of drawbacks, as identified by Bannò *et al.* [16]. First, they argue that a framework built on top of an ad-hoc modified JVM *becomes less portable*, since it could not be used with any other standard virtual machines. Moreover, *new versions of code that is loaded dynamically may bypass security checks performed by the JVM* (no other arguments are provided) and *code optimisations executed by the JVM at reunite may modify the internal structure and the flow of operations of the application, thus making changes difficult to be properly applied*. Actually, there are some other reasons. For instance, the update mechanism becomes dependent on a specific version of the JVM. To keep the mechanism updated, the developers would have to modify any new version of the JVM that were released and check that the mechanism goes on working properly, which represents a significant effort. Moreover, such new versions of the JVM may suffer changes in their design or implementation that prevent the required modifications from being applied, thus yielding the update mechanism outdated in short period of time.

Finally, as a particular case of Java technology, the OSGi standard offers a *standard* mechanism to dynamically reload the bundles that compose an OSGi application (see Section 3.6.3 for additional references about OSGi).

3.6.3 OSGi

OSGi [6, 13] 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 (for instance, 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*, in the standard `a.b.c` Java package convention. 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 `a.b.c`-like package list and optionally, for each package, the minimal version that is required. This expresses the dependencies the bundle relies on, including general packages included in 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 version of the bundle can be reloaded, by means of a mechanism provided by OSGi. Then OSGi keeps the old version of the bundle available to those bundles that already had a *reference* to it (in order to let them progress correctly) and offers the new version of the bundle to those bundles that get a reference to the bundle, from that moment on.

This can be explained easily with a simple example. Consider two bundles, A and B, that depend on a bundle Z. Then consider the following sequence of actions and events. First A is loaded and started. A is able to know that no implementation of Z is available and waits. Then Z is loaded and started. A is informed about and asks to OSGi for a *reference* to Z. Then Z is updated. As there are at least one bundle (A) referencing to the old version of Z, OSGi keeps both old and new versions of Z and A keeps its reference to the *old version* of Z. Then B is started and asks to OSGi for a *reference* to Z. In this case, B gets a reference to the *new version* of Z.

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 [69]. Moreover, there are a number of implementations of OSGi, like Apache Felix [30], Concierge [1, 60] (especially designed for resource-constrained devices), Equinox [2], KnopflerFish [3] and Oscar [5], 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 [59], an extension of the standard OSGi specification to build distributed systems. Another alternative also focused on distributed and cloud systems is OSGi*4C* [62]. And in [26], Chen and Huang propose a mechanism to dynamically update the bundles of an OSGi application.

3.6.4 Dynamic software update in the .NET platform

In this section we survey some options related to dynamic software update that are available for .NET applications.

First, there are two technologies included in Microsoft .NET 4 platform that allow to dynamically *load* code. These are the *Managed Extensibility Framework (MEF)* and the *Managed Add-In Framework* (MAF).

The Managed Extensibility Framework (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, in MEF, each 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 assemblies are run. Classes in different application domains can not communicate directly (by means of a regular local invocation),

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*.

The Managed Add-In Framework (MAF) is a technology similar to MEF although there are some differences. Regarding DSU, the most important difference is that it 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 [28], 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 in 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 can not 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 is used to load 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 can not 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 can not 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 can not 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 penalty performance 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. The main issue is the *inability* of unloading individual assemblies from a given application domain. The second issue is the need to use slow IPC-like mechanisms to communicate classes belonging to different application domains.

3.6.5 Dynamic software update in Erlang

Erlang [10] 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

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 referred to above.

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

3.6.6 Version Coexistence

Version coexistence is the ability of a dynamic update system to allow different versions of an updateable component to concurrently coexist, providing a regular service according to their specifications. In Section 2 we identified this feature as one of the fundamental requirements a dynamic update mechanism should have. However, the support needed to provide it may have a cost, from different points of view. First, it has to be implemented, which means a significant effort. Then, it may have some other cost in run-time, imposing some performance overhead regarding an update mechanism without such a support.

Thus, in many of the proposals reviewed, 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 3.1), performing the update and *uninstalling* or otherwise preventing both versions to run at the same time.

Nevertheless, there are some authors that provide some support to version coexistence. For instance, in the context of *dynamic updating* of functions and procedures, Segal and Frieder [63, 32], 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.* [12, 11] 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 [24] and [25, Section 2.2] allows 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*.

Dmitriev [27] elaborates on different policies that may be implemented in the context of the dynamic update mechanism included in the Java Virtual Machine.

3.6.7 Replication

Few papers have tackled the topic of applying dynamic updates to replicated systems. For instance, Solarski and Meling [65] 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 [64] in his Ph.D. Thesis, by adding a procedure applicable to systems that use *passive replication*. The procedure does not actually apply a dynamic update of the replicas. Instead, the new version of the software is installed in brand new replicas and the old ones are shut down manually. Finally, a failover mechanism is used to promote to primary replica one of the new replicas.

Wang *et al.* [73] propose a mechanism to dynamically change the consistency mode used by the replicas of a replicated system. They argue that the consistency needs of a replicated system can change in run-time, during the regular execution, according to the observed rates of read and write operations issued by the clients. The rate of read an write operations may be *low* or *high* and thus, at any moment, the system can be in any of the four possible combinations. The authors propose to consider the current combination

to dynamically change the consistency mode of the replicas, from a *relaxed* consistency mode to a *strong* consistency mode. They also organize 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 first combination is a *strong consistency mode* in which the read rate is high and the write rate is low. Write operations sent to any replica are redirected to the master, which redirects them to all the replicas, thus achieving full consistency among all the replicas. Read requests are sent to any replica and can attended immediately because they are all updated.

The second combination is a *trade-off mode* in which both the read and write rates are high. Write operations sent to any replica are redirected to the master. The master forwards them to the deputy nodes. It also forwards them to the rest of the replicas if it considers that they are too outdated. Read requests are sent to any replica and attended immediately (so an *old* value may be read).

The third combination is another *trade-off mode* in which the read rate is low and the write rate is high. Write operations sent to any replica are resent to the master, who resends them to all the deputy nodes. Read requests sent to the master or to any of the deputy nodes are responded immediately. Read requests sent to a child node are forwarded to and answered by the *closest* deputy thus ensuring that an updated value is provided.

The fourth combination is also a *trade-off-mode* in which both the read and write rates are low. The only different case is when a child node receives a read request. If the child it *too outdated*, it retrieves the requested value from the closest deputy node and returns it to the client. Otherwise, the child node answers itself (so an *old* value may be read).

3.6.8 Scheduling and synchronizing

In Section 3.4 we provided a number of references of systems that allow the user to mark places in the program. In some cases, those are places in which a dynamic update may be applied. In other cases, they are places in which a dynamic update should not be applied.

Other systems consider the scheduling of the updates at a higher level of abstraction. For instance, Ajmani *et al.* [12, 11] propose the use of *scheduling functions*, in the context of updating distributed systems. 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 [21]) that may be implemented as *scheduling functions*. For instance, a *fast reboot* update consists in updating all nodes at once. In general, this is considered a *bad option*, since it yields the software system completely unavailable during the time required by the update to take place. 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 need to be compatible since they will coexist.

They also identify other types of *patterns*, used to *synchronize* the progression of the nodes. For instance, one pattern may be *to wait until all nodes are updated*, which can be seen as some sort of *strong synchronization barrier* among all nodes. A less restrictive option may be *to wait until all nodes of class C are updated*. Moreover, the load of the nodes can also be considered so there can be a pattern *to wait until the node is lightly loaded* (with some definition of *load* and some criterion to measure it).

For those cases in which some *global knowledge* of the state of the nodes is needed, they provide a *central upgrade database* component that gathers and spreads information about the state of the nodes.

3.6.9 Rollbacks

Some mechanisms offer the possibility to *rollback* or *undo* an update.

For instance POLUS [24, 25] uses a mechanism based on the generation of *dynamic patches* to update a running program from version v to $v + 1$. The mechanism can also be applied to *rollback* an update, for

instance when it is not behaving correctly or for any other reason, as decided by the programmer. For this, it is enough to apply a patch to *update* the program from version $v + 1$ to v.

POLUS uses a carefully designed *indirection* mechanism that avoids multiple indirections. When a function is updated, from a version v to $v + 1$, by means of a *dynamic patch*, POLUS inserts a *jump* instruction to redirect to the proper version implementation the requests made to the function. 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 patches. Besides applying the patches, POLUS also *undoes* the insertion of the corresponding *jump* instructions, thus again avoiding unnecessary *back and forth* redirections.

In [22], Brown and Patterson propose a model for rollback mechanisms, as a solution to the *external inconsistency* problem. This happens when the rollback of an update made to an application also discards changes to the data that have been *seen* by the user.

The proposed model is based on three stages or steps: *rewind*, *repair* and *replay*. In the *rewind* step, the rollback mechanism rollbacks 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, *over* the rollbacked 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[3].

The proposed model presents a *lack of genericity* problem, since it depends on particular *low level* 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.

The authors have implemented a prototype that is usable to rollback updates to regular email client applications that use the IMAP and SMTP protocols. They do not provide any details about how the updates are first applied and then rollbacked.

4 Discussion

The *dynamic software update* topic has been being studied for decades (the first well-known references are from the 70s), in the context of both *centralized* and distributed systems. Since then, many papers have been published and there exist many practical proposals. Some of them are for specific contexts and situations but others were designed to be generally applied and used. For instance, there exist proposals for the dynamic update of software written in current languages like C or Java.

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, many applications require the restart of the browser in order to update it. First, the application itself detects the existence of a new available update and downloads it to the local filesystem. Next, the user is asked to restart the application so the update can be applied during the next restart of the

[3] In a certain sense, the difference between using a concrete low level format or a semantic one is similar to the difference found in a similar situation, in the context of replicated databases. When a replica node needs to send to other node some changes made to a replicated database it can send a *writeset* with the changes or a representation of the corresponding *operation* (for instance, a regular `UPDATE` SQL sentence). In this example, some other criteria have to be considered, like the *length* of the writeset or the *computational cost* of re-executing the SQL sentence in the destination replica.

browser (the user can usually postpone this action). Once the application has been restarted and the update applied, the user can resume using it normally.

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 restart of the system. And in some other cases, updates are dynamically applied but require the restart of the system for the user to use 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 the one hand there are a number of *big* web applications (that usually belong to *big* 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 at both the server level and the database level which allows the updates to be applied transparently so the users do not usually realize when the updates are applied.

On the other hand, there is the case of *small* web applications, typically belonging to *smaller* entities that do not have the same availability of resources of the *big 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 information 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 the drawbacks pointed out in Section 1, the applications and servers to update should have some dynamic update mechanisms that allowed them to be updated 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 the users and the holding entity or company.

We must also consider the case of *cloud computing* infrastructures. Specifically, the use of a dynamic update mechanism, transparent to the users, makes even more sense in the case of the applications that are executed in those infrastructures.

Indeed, one of the goals of any cloud infrastructure is to allow the applications to which it gives support, to continuously run so they can continuously provide their services to their users without any interruption. For this, the own infrastructures must also 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* [72]. In all these cases, there are a number of software components that may also need to be dynamically updated (to fix bugs and security issues, add new features, etc.). 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 cases of IaaS, the user has some *virtualized environment* in which the user can *drop* applications. In case of PaaS providers, the offered service consists in a set of tools, utilities, libraries, etc. that can be used by the users to 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 network (typically, through a regular web browser) and used like any other locally installed application. Cloud providers may include dynamic update mechanisms so their own cloud infrastructures can be dynamically updated.

On the other hand, the user applications that are executed in IaaS and PaaS cloud infrastructures can 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 infrastructure. For instance, applying an update to a cloud-hosted application could be as easy as applying the update in a *static* manner, asking for new *instances* once the application has been completely updated and, in parallel, asking for the halt of existing outdated versions of the application. In the particular case of applications that have been specifically designed to be run in cloud environments, the designers and developers may know the problems that may arise and may have taken some precautions by including specific mechanisms (for instance, mechanisms to synchronize the instances of the application and merge their states).

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 we discuss some *issues* related with those.

Issues related to *low-level* techniques. Some techniques are applied at a low level of abstraction or depend upon low level of abstraction details. For instance, some techniques are based on the inspection of the execution program stack (see Section 3.1), 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 3.1. 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 (for instance, fragments of a function), directly altering the instructions loaded in the 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 (an others based on the Java Virtual Machine like Scala, Groovy and others), C# (and other languages that depend on the .NET platform), Perl, PHP, Python, Ruby, Erlang and many others. 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 still use some binary rewriting techniques (see Section 3.2) 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 update 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 language 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 low level technique) and at the same time is easily portable to different platforms.

Finally, we must consider that the use of *high levels of abstraction* eases the use of other underlying abstractions. For instance, in the context of cloud computing systems, many cloud infrastructure providers offer abstractions based on the use of *virtual machines* and other *virtualization* mechanisms. The use of low level techniques like the ones discussed above may *obstruct* (and in a worst case, even prevent) the update mechanism and the cloud infrastructure itself to properly work.

For all these reasons, we discourage the use of binary rewriting techniques in particular and any other technique of a low level of abstraction is not a recommendable option when high availability is a requirement.

Issues related to source code analysis. Other techniques are based on some analysis (typically static) of the source code of the updateable applications. For instance, in some cases, such an analysis is used to identify points in the updateable programs in which a dynamic update may or *may not* be applied. The advantage of these techniques is that the programmer does not need to worry about such tasks thus improving the *programmer transparency*.

Nevertheless, it must be considered that these techniques can only perform these analyses based on the structure of the source code (this is, only from a *syntactic* point of view). The problem is that, on practice, there may be semantic details that may remain unnoticed for the analyzers. For instance, the program to update may use some *concept* of *transaction* that should be respected. The syntactic analysis of the program may identify *safe* or *update points* in which dynamic updates may be applied but it may happen that such an update interrupted the execution of a transaction. A part of the transaction may be executed with the old version of a given function and the rest may be executed with the updated version of another function, which could lead the program to an undesirable state.

As we discuss later, rather than depending on a static analysis of the source code we recommend a more comprehensive analysis performed by the developers of the program. The advantage of this approach is that it allows to take better decisions, since the developers have a complete *semantic* view of the program.

Issues related to the use of *intermediary levels*. The use of intermediary levels also 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 intermediary levels 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. This means that this overhead must be carefully estimated first and then measured.

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 have also 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*. In principle, the concept of *version coexistence* offers some advantages, especially combined with other techniques and approaches (like the use of intermediary levels).

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, we may need to ensure that they 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 source, physical resources, etc. In any case, the support to version coexistence leads to complications that must be evaluated.

Issues related to replication. We have found few references that tackle the problem of dynamic update in the case of replicated (distributed) systems.

Among them, one proposes a couple of procedures to dynamically update actively and passively replicated systems. The procedure for actively replicated systems relies on a group communication system offering a total order message delivery service. This procedure does not actually use any of the techniques discussed in Section 3, besides some state transfer step, used to update the new replicas as they are started. Instead, a regular *stop-and-update* procedure is followed, to update, one by one, all the replicas. The procedure for passively replicated systems does not use any of those techniques, too. Instead, it relies on the regular *replica failover* mechanism that may be part of the replication protocol.

The problem of these procedures is that they depend on a *state transfer* step to update the new replicas. As the state to transfer may be large, this step may be too *expensive* in terms of the time needed to perform it. Moreover, it may be needed to block the replicated system during the state transfer step.

Necessity of *intrusion*. One of the main conclusions we can draw is that the dynamic software update problem is complex enough for a tool to solve it in a completely transparent manner, especially from the point of view of the final user. Moreover, we find that some of the goals listed in Section 2 are conflicting. For instance, the *user transparency* and *programmer transparency* goals are somehow conflicting.

As pointed out in Section 3.4, a number of authors propose the use of *intrusive* mechanisms and techniques, in order to get a functional dynamic update solution, even if not all the desirable requirements are fulfilled.

In this sense, one of the techniques that can be used to achieve such a goal is the use of marks and annotations in the source code to identify *safe* or *update* points, in which a dynamic update can take place. This technique may be preferable to *automated* techniques based on the analysis of the source code, because the programmer of the updateable program is the one that can better choose such places. The choice can be done considering semantic constraints (like the presence of transactions)

The programmer can also help to ensure that the program or component is kept in any sort of *quiescent* state, if needed. For instance, *idle functions* may be provided and executed to help the update mechanism to ensure that no relevant code is executed during an update.

The programmer may also provide the functionality required to transform and transfer the state of the updateable components or programs. Again, the programmer is the one that better knows how the program manages its state, how has it to be represented, and how should it be transformed.

The previous techniques can be combined with the use of some sort of *underlying support*. This can be some kind of *middleware*, *framework* or *library*. A good option available in the Java *universe* is the use of an OSGi implementation (see Section 3.6.3).

5 Conclusion

This report surveys a number of references related to the *dynamic software update* topic. The main goal is to introduce the topic to the interested readers in a *structured* manner and help them to learn about a number of references available in the literature of this topic. The report offers a twofold contribution.

First, we study the variety of *definitions* of *dynamic software update* found in the surveyed references. In Section 2 we provide a selection of the most important requirements chosen by the authors.

Then, we also analyze which are the techniques and other related concepts and issues in those references and identify which are the most used. The corresponding selection can be found in Section 3.

Finally, in Section 4 we discuss the techniques described in Section 3.

References

[1] Concierge. http://concierge.sourceforge.net/.

[2] Equinox. http://eclipse.org/equinox/.

[3] Knopflerfish. http://www.knopflerfish.org/.

[4] Open Source ByteCode Libraries in Java. http://java-source.net/open-source/bytecode-libraries.

[5] Oscar. http://oscar.objectweb.org.

[6] OSGi Alliance. http://www.osgi.org.

[7] CGLib 2.2.2, April 2011. http://cglib.sourceforge.net/.

[8] Javeleon 1.5, September 2011. http://javeleon.org.

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

[10] Ericsson AB. Erlang. http://www.erlang.org.

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

[12] Sameer Ajmani, Barbara Liskov, and Liuba Shrira. Modular Software Upgrades for Distributed Systems. In *European Conference on Object-Oriented Programming (ECOOP)*, July 2006.

[13] OSGi Alliance. About the OSGi Service Platform. Technical Whitepaper. Revision 4.1., June 2007.

[14] Joao Paulo Almeida, Marteen Wegdam, Marten van Sinderen, and Lambert Nieuwenhuis. Transparent Dynamic Reconfiguration for CORBA. In *3rd International Symposium on Distributed Objects and Applications (DOA)*, pages 197–207, 2001.

[15] Gautam Altekar, Ilya Bagrak, Paul Burstein, and Andrew Schultz. OPUS: Online Patches and Updates for Security. In *14th Conference on USENIX Security Symposium*, SSYM'05, Baltimore, MD, 2005. USENIX Association.

[16] Filippo Bannò, Daniele Marletta, Giuseppe Pappalardo, and Emiliano Tramontana. Handling Consistent Dynamic Updates on Distributed Systems. In *2010 IEEE Symposium on Computers and Communications (ISCC)*, pages 471–476, June 2010.

[17] Mario R. Barbacci, Dennis L. Doubleday, Charles B. Weinstock, Michael J. Gardner, and Randall W. Lichota. Building Fault Tolerant Distributed Applications with Durra. In *International Workshop on Configurable Distributed Systems*, pages 128–139, March 1992.

[18] Jaroslav Bauml and Premek Brada. Automated Versioning in OSGi: a Mechanism for Component Software Consistency Guarantee. In *35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '09)*, pages 428–435, August 2009.

[19] Christophe Bidan, Valérie Issarny, Titos Saridakis, and Apostolos Zarras. A Dynamic Reconfiguration Service for CORBA. In *Fourth International Conference on Configurable Distributed Systems*, pages 35–42, May 1998.

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

[21] Eric A. Brewer. Lessons from Giant-Scale Services. *IEEE Internet Computing*, 5(4):46–55, July 2001.

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

[23] Eric Bruneton, Romain Lenglet, and Thierry Coupaye. ASM: a Code Manipulation Tool to Implement Adaptable Systems. November 2002.

[24] Haibo Chen, Jie Yu, Rong Chen, Binyu Zang, and Pen-Chung Yew. POLUS: A POwerful Live Updating System. In *29th international conference on Software Engineering*, ICSE '07, pages 271–281. IEEE Computer Society, May 2007.

[25] Haibo Chen, Jie Yu, Chengqun Hang, Binyu Zang, and Pen-Chung Yew. Dynamic Software Updating Using a Relaxed Consistency Model. *IEEE Transactions on Software Engineering*, 37(5):679–694, September-October 2011.

[26] Junqing Chen and Linpeng Huang. Dynamic Service Update Based on OSGi. In *WRI World Congress on Software Engineering (WCSE '09)*, volume 3, pages 493–497, Xiamen, China, May 2009. IEEE Computer Society.

[27] M. Dmitriev. Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications. In *Workshop on Engineering Complex Object-Oriented Systems for Evolution, in association with OOPSLA 2001 International Conference*, 2001.

[28] Clément Escoffier, Didier Donsez, and Richard S. Hall. Developing an OSGi-like Service Platform for .NET. In *IEEE Consumer Communications and Networking Conference (CCNC'06)*, volume 1, pages 213–217, January 2006.

[29] R. S. Fabry. How to Design a System in Which Modules Can be Changed on the Fly. In *2nd International Conference on Software Engineering (ICSE '76)*, pages 470–476, San Francisco, California, United States, 1976. IEEE Computer Society Press, Los Alamitos, CA, USA.

[30] The Apache Software Foundation. Apache Felix. http://felix.apache.org.

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

[32] Ophir Frieder and Mark E. Segal. On Dynamically Updating a Computer Program: from Concept to Prototype. *Journal of Systems and Software*, 14(2):111–128, February 1991.

[33] Cristiano Giuffrida and Andrew S. Tanenbaum. A Taxonomy of Live Updates. In *Advanced School for Computing and Imaging (ASCI) 2010 Conference*, Veldhoven, The Netherlands, November 2010.

[34] James Gosling, Bill Joy, Guy Steele, and Gilad Bracha. *The Java Language Specification*. Third edition edition, 2005.

[35] Allan Raundahl Gregersen and Bo Nørregaard Jørgensen. Dynamic Update of Java Applications—balancing Change Flexibility vs Programming Transparency. *Journal of Software Maintenance and Evolution: Research and Practice*, 21(2):81–112, March 2009.

[36] Allan Raundahl Gregersen, Douglas Simon, and Bo Nørregaard Jørgensen. Towards a Dynamic-update-enabled JVM. In *Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, Genova, Italy, 2009. ACM.

[37] Allan Raundahl Gregersen, Douglas Simon, and Bo Nørregaard Jørgensen. Towards a Dynamic-update-enabled JVM. In *Workshop on AOP and Meta-Data for Software Evolution*, RAM-SE '09, Genova, Italy, 2009. ACM.

[38] Deepak Gupta. *On-line Software Version Change*. PhD thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, India, November 1994.

[39] Deepak Gupta and Pankaj Jalote. On Line Software Version Change Using State Transfer Between Processes. *Software Practice and Experience*, 23(9):949–964, September 1993.

[40] Deepak Gupta, Pankaj Jalote, and Gautam Barua. A Formal Framework for On-line Software Version Change. *IEEE Transactions on Software Engineering*, 22(2):120–131, February 1996.

[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, 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.

[43] Michael Hicks and Scott Nettles. Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 27(6):1049–1096, November 2005.

[44] Christine R. Hofmeister and James M. Purtilo. A Framework for Dynamic Reconfiguration of Distributed Programs. Technical Report UMIACS-TR-93-78, 1993.

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

[46] J. Kramer, J. Magee, M. Sloman, and A. Lister. CONIC: an Integrated Approach to Distributed Computer Control Systems. *IEE Proceedings E Computers and Digital Techniques*, 130(1), January 1983.

[47] Jeff Kramer and Jeff Magee. Dynamic Configuration for Distributed Systems. *IEEE Transactions on Software Engineering*, SE-11(4):424–436, April 1985.

[48] Jeff Kramer and Jeff Magee. The Evolving Philosophers Problem: Dynamic Change Management. *IEEE Transactions on Software Engineering*, 16(11):1293–1306, Nov. 1990.

[49] Eugene Kuleshov. Using ASM Framework to Implement Common Bytecode Transformation Patterns. Vancouver, Canada, March 2007.

[50] Tim Lindholm and Frank Yellin. The Java Virtual Machine Specification, Second Edition, 1999.

[51] Barbara Liskov and Robert Scheifler. Guardians and Actions: Linguistic Support for Robust, Distributed Programs. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 5(3):381–404, July 1983.

[52] Valerio Panzica La Manna. Dynamic Software Update for Component-based Distributed Systems. In *Proceedings of the 16th international workshop on Component-oriented programming*, WCOP '11, pages 1–8, New York, NY, USA, 2011. ACM.

[53] Marco Milazzo, Giuseppe Pappalardo, Emiliano Tramontana, and Giuseppe Ursino. Handling Runtime Updates in Distributed Applications. In *2005 ACM symposium on Applied computing*, SAC '05, pages 1375–1380, Santa Fe, New Mexico, 2005. ACM.

[54] Yogesh Murarka and Umesh Bellur. Correctness of Request Executions in Online Updates of Concurrent Object Oriented Programs. In *15th Asia-Pacific Software Engineering Conference (APSEC '08)*, pages 93–100. IEEE Computer Society, December 2008.

[55] Iulian Neamtiu, Michael Hicks, Gareth Stoyle, and Manuel Oriol. Practical Dynamic Software Updating for C. In *ACM SIGPLAN conference on Programming language design and implementation*, PLDI '06, pages 72–83, Ottawa, Ontario, Canada, 2006. ACM.

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

[57] James M. Purtilo. The POLYLITH Software Bus. *ACM Transactions on Programming Languages and Systems*, 16(1):151–174, January 1994.

[58] James M. Purtilo and Christine R. Hofmeister. Dynamic Reconfiguration of Distributed Programs. In *11th International Conference on Distributed Computing Systems*, pages 560–571, May 1991.

[59] Jan Rellermeyer, Gustavo Alonso, and Timothy Roscoe. R-OSGi: Distributed Applications Through Software Modularization. In Renato Cerqueira and Roy Campbell, editors, *Middleware*, volume 4834 of *Lecture Notes in Computer Science*, pages 1–20, Newport Beach, CA, USA, 2007. Springer Berlin, Heidelberg.

[60] Jan S. Rellermeyer and Gustavo Alonso. Concierge: a Service Platform for Resource-constrained Devices. In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems*, volume 41 of *EuroSys '07*, pages 245–258, Lisbon, Portugal, March 2007. ACM.

[61] Tobias Ritzau and Jesper Andersson. Dynamic Deployment of Java Applications. In *Java for Embedded Systems Workshop*, London, United Kingdom, May 2000.

[62] Holger Schmidt, Jan-Patrick Elsholz, Vladimir Nikolov, Franz J. Hauck, and Rüdiger Kapitza. OSGi4C: Enabling OSGi for the Cloud. In *Fourth International ICST Conference on COMmunication System softWAre and middlewaRE (COMSWARE '09)*, COMSWARE '09, Dublin, Ireland, June 2009. ACM.

[63] Mark E. Segal and Ophir Frieder. Dynamic Program Updating in a Distributed Computer System. In *Conference of Software Maintenance*, pages 198–203, Scottsdale, AZ, USA, October 1988.

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

[65] Marcin Solarski and Hein Meling. Towards Upgrading Actively Replicated Servers on-the-fly. In *26th Annual International Computer Software and Applications Conference (COMPSAC 2002)*, pages 1038–1043, 2002.

[66] N. Sridhar, S.M. Pike, and B.W. Weide. Dynamic Module Replacement in Distributed Protocols. In *23rd International Conference on Distributed Computing Systems*, pages 620–627, May 2003.

[67] Gareth Stoyle, Michael Hicks, Gavin Bierman, Peter Sewell, and Iulian Neamtiu. Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Transactions on Programming Languages and Systems (TOPLAS)*, 29(4), August 2007.

[68] Michiaki Tatsubori, Toshiyuki Sasaki, Shigeru Chiba, and Kozo Itano. A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In *15th European Conference on Object-Oriented Programming (ECOOP '01)*, ECOOP '01, pages 236–255. Springer-Verlag, 2001.

[69] Andre L. C. Tavares and Marco Tulio Valente. A Gentle Introduction to OSGi. *SIGSOFT Software Engineering Notes*, 33(5), September 2008.

[70] L.A. Tewksbury, L.E. Moser, and P.M. Melliar-Smith. Live Upgrades of CORBA Applications Using Object Replication. In *Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01)*, ICSM '01, pages 488–497. IEEE Computer Society, 2001.

[71] Yves Vandewoude, Peter Ebraert, Yolande Berbers, and Theo D'Hondt. Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *IEEE Transactions on Software Engineering*, 33(12):856–868, December 2007.

[72] Luis M. Vaquero, Luis Rodero-Merino, Juan Caceres, and Maik Lindner. A Break in the Clouds: owards a Cloud Definition. *SIGCOMM Computer Communication Review*, 39(1):50–55, January 2009.

[73] Ximei Wang, Shoubao Yang, Shuling Wang, Xianlong Niu, and Jing Xu. An Application-Based Adaptive Replica Consistency for Cloud Storage. In *2010 Ninth International Conference on Grid and Cooperative Computing*, pages 13–17, Nanjing, November 2010.

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