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

Abstract. Software systems are continuously evolving. New features are requested and then added and bugs are found and then fixed. The drawback of a classic *stop, update and restart* model is that it reduces the availability of the software. A *Dynamic Software Update* (DSU) mechanism allows to dynamically apply updates to running software, without having to stop it. This paper is a short survey on the existing literature related to DSU. We present a selection of the main *goals and requirements* of a DSU mechanism, as identified by a number of authors. We also provide a selection of the most common techniques and issues considered in the surveyed references.

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, 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

* This work has been supported by EU FEDER and Spanish MICINN under research grants TIN2009-14460-C03-01 and TIN2010-17193.

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. The paper is concluded in Section 4. An extended version of this survey can be found in [50].

2 Requirements and Goals

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 references that identify it explicitly are [26,61,29,62,52,32].

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.

Many authors consider a *relaxed version* of this requirements. 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 ([45,61,29,40,22,32,49]). In other cases, this requirement is more specific, like in [26], which admits a *momentary delay* in the normal execution of user requests or [62] 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). Several types of transparency can be considered.

The *user transparency* is 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.

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.

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, [32] requires *programmer transparency* and [62,15] require *application transparency*.

Generality. The update process is general. 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 [10,11] and [32] while the second interpretation is used by [62]. Moreover, [49] 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 [45,64,62,52] 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*. [32] is a bit more specific and requires that the state of the software after a dynamic update 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. [15] requires *data consistency* and also *consistency of flow* (the proper termination of pending requests). Finally, [49] 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 persistent 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 references declare this requirement explicitly, like [64,10,11] 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 references, like [10,11] (they call it *mixed mode operation*) and [62] include a requirement similar to this one.

Other Requirements. In [50] we include some additional secondary requirements (*Atomicity and Rollbackness*, *Schedulability and Automation* and *Simultaneous Updates*).

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 [37,36,61,29,56,42,55] 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.).

Reach of a Safe Point. Some techniques [37,24,30] 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.

Communication Quiescence. The original concept of *quiescence* was defined by [45] 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 [45] 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, [69] argues that the *quiescence* concept in [45] 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. In

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

[50] we provide a short comparison between the original *quiescence* and the *tranquility* concepts.

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 [15] in their FREJA framework.

Other References. This idea of *stable status* or *quiescence* appears in many other references: [19,16,41,18,59,13,68,39]. 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, [25] 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.

This was one of the first techniques proposed to be used by a dynamic update mechanism. Nevertheless, it has a number of disadvantages. First, it 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. Moreover, it is

also difficult to port to other architectures and would force the programmers to have a deep low level knowledge of both the *source* and the *target* architectures.

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 references to propose the use of *binary redirection* was [26]. 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.

In [26] two different alternatives to perform such a redirection are proposed. 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.

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.

In [40], they use some binary rewriting techniques to modify the service implementation, data types and the client code that accesses to the patched code. 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.

In [22,23], the authors 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 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).

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 ([31,47]). The modifications are typically expected to preserve the *Java binary compatibility* ([31]). As in the previous cases, this technique also has the disadvantage of depending upon

a binary level although in this case, it has a minor practical impact, since the Java language is widely supported by many operating systems and hardware platforms.

Several authors have studied the use of binary rewriting in Java programs [51,32,15]. 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 [54,21,46], CGLIB [7], Javassist [9,66], Apache Commons BCEL [28], Javeleon [8,33], JRebel [71] 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).

A number of authors [26,19,61,29,22,23,56,55,64,51,11,32,24,15] have used different versions of this approach. They are briefly reviewed in [50].

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 sucessfully finished. The update mechanism proposed by [45] is one of the first works that follows this approach.

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).

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. Different versions of this approach are considered by [29,53,40,15,30] and briefly described in [50].

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 [38]. 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.

A number of authors [19,56,55,36,63,62,64,32,24,15] have considered this technique. 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 [19,29,56,40,11,65,52,24].

3.6 Other Issues

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

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.

In [50] we discuss a number of proposals that use some sort of static analysis of the source code [65,52,53,14,40,22,23,17].

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, [61,29] need that the hardware architecture of the underlying machine offers an *indirect addressing mode*. The proposal in [36] 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. Other references propose solutions that are a bit more general and can be used with programs written in *imperative languages*, like [40]. There are also some authors who develop their proposal based on their own infrastructure. For instance, [44,45] is based on their CONIC configuration language and infrastructure [43]. In Proteus, the authors [65] describe a dynamic software update solution based on its own programming language, compiler and run-time, among other tools and resources. In [19] a dynamic update solution for programs written with the Argus programming language ([48]) is presented. For the C programming languages there are some options, like [35,53,22,23]. Regarding the Java programming language and Virtual Machine (JVM), there are a large number of references [59,51,32,15,25,34].

As a particular case of Java technology, the OSGi platform [6,12] allows to build Java applications from a number of modular, reusable and collaborative components (called *bundles*), that can be dynamically reloaded. The applications register their bundles in an OSGi server (or implementation), that acts as a software bus, thus allowing a loosely coupling among the bundles. These can be dynamically reloaded and OSGi provides some sort of *snapshot* view of the bundles. When a bundle is reloaded, existing bundles keep an *old-version view* of it while bundles started from then on just *see* the new version. In [50] we provide a more detailed explanation of this and other features of OSGi. A short introduction to OSGi can be found in [67]. Moreover, there are a number of implementations of OSGi, like Apache Felix [27], Concierge [1,58] (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 it in some way. R-OSGi [57] is an extension of the standard OSGi specification to build distributed systems. OSGi*4C* [60] focuses on distributed and cloud systems and [24] includes a mechanism to dynamically update the bundles of an OSGi application.

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 [61,29,11,10,22,23,25].

Replication Few papers have tackled the topic of applying dynamic updates to replicated systems. For instance, [63] proposes 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 [62], 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.

Moreover, [70] proposes 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.

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, [11,10] 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.

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

For instance POLUS [22,23] 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.

Moreover, in [20], the authors propose a model for rollback mechanisms, as a solution to the *external inconsistency* problem, which happens when the rollback of an update made to an application also discards changes to the data that have *seen* by the user. An extended explanation can be found in [50].

4 Conclusion

This report is a reduced version of a technical report in which we survey 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. 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, in Section 3, we also analyze which are the techniques and other related concepts and issues in those references and identify which are the most used.

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. Ajmani, S.: Automatic Software Upgrades for Distributed Systems. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (2004)
11. Ajmani, S., Liskov, B., Shrira, L.: Modular Software Upgrades for Distributed Systems. In: European Conference on Object-Oriented Programming (ECOOP) (July 2006)
12. Alliance, O.: About the OSGi Service Platform. Technical Whitepaper. Revision 4.1. (June 2007)
13. Almeida, J.P., Wegdam, M., van Sinderen, M., Nieuwenhuis, L.: Transparent Dynamic Reconfiguration for CORBA. In: 3rd International Symposium on Distributed Objects and Applications (DOA). pp. 197–207 (2001)
14. Altekar, G., Bagrak, I., Burstein, P., Schultz, A.: OPUS: Online Patches and Updates for Security. In: 14th Conference on USENIX Security Symposium. SSYM'05, USENIX Association, Baltimore, MD (2005)

15. Bannò, F., Marletta, D., Pappalardo, G., Tramontana, E.: Handling Consistent Dynamic Updates on Distributed Systems. In: 2010 IEEE Symposium on Computers and Communications (ISCC). pp. 471–476 (June 2010)
16. Barbacci, M.R., Doubleday, D.L., Weinstock, C.B., Gardner, M.J., Lichota, R.W.: Building Fault Tolerant Distributed Applications with Durra. In: International Workshop on Configurable Distributed Systems. pp. 128–139 (March 1992)
17. Bauml, J., Brada, P.: Automated Versioning in OSGi: a Mechanism for Component Software Consistency Guarantee. In: 35th Euromicro Conference on Software Engineering and Advanced Applications (SEAA '09). pp. 428–435 (August 2009), `http://ieeexplore.ieee.org/assets/img/btn.pdf-access-full-text.gif`
18. Bidan, C., Issarny, V., Saridakis, T., Zarras, A.: A Dynamic Reconfiguration Service for CORBA. In: Fourth International Conference on Configurable Distributed Systems. pp. 35–42 (May 1998)
19. Bloom, T.: Dynamic Module Replacement in a Distributed Programming System. Ph.D. thesis, Massachusetts Institute of Technology, Cambridge, MA, USA (1983)
20. Brown, A.B., Patterson, D.A.: Rewind, repair, replay: three R's to dependability. In: 10th workshop on ACM SIGOPS European workshop. pp. 70–77. EW 10, ACM, Saint-Emilion, France (2002)
21. Bruneton, E., Lenglet, R., Coupaye, T.: ASM: a Code Manipulation Tool to Implement Adaptable Systems (November 2002)
22. Chen, H., Yu, J., Chen, R., Zang, B., Yew, P.C.: POLUS: A POwerful Live Updating System. In: 29th international conference on Software Engineering. pp. 271–281. ICSE '07, IEEE Computer Society (May 2007)
23. Chen, H., Yu, J., Hang, C., Zang, B., Yew, P.C.: Dynamic Software Updating Using a Relaxed Consistency Model. IEEE Transactions on Software Engineering 37(5), 679–694 (September-October 2011)
24. Chen, J., Huang, L.: Dynamic Service Update Based on OSGi. In: WRI World Congress on Software Engineering (WCSE '09). vol. 3, pp. 493–497. IEEE Computer Society, Xiamen, China (May 2009)
25. Dmitriev, M.: 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)
26. Fabry, R.S.: How to Design a System in Which Modules Can be Changed on the Fly. In: 2nd International Conference on Software Engineering (ICSE '76). pp. 470–476. IEEE Computer Society Press, Los Alamitos, CA, USA, San Francisco, California, United States (1976)
27. Foundation, T.A.S.: Apache Felix, http://felix.apache.org
28. Foundation, T.A.S.: Apache Commons BCEL 6.0 (October 2011), http://commons.apache.org/bcel/
29. Frieder, O., Segal, M.E.: On Dynamically Updating a Computer Program: from Concept to Prototype. Journal of Systems and Software 14(2), 111–128 (February 1991)
30. Giuffrida, C., Tanenbaum, A.S.: A Taxonomy of Live Updates. In: Advanced School for Computing and Imaging (ASCI) 2010 Conference. Veldhoven, The Netherlands (November 2010)
31. Gosling, J., Joy, B., Steele, G., Bracha, G.: The Java Language Specification. Third edition edn. (2005)
32. Gregersen, A.R., Jørgensen, B.N.: 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)
33. Gregersen, A.R., Simon, D., Jørgensen, B.N.: Towards a Dynamic-update-enabled JVM. In: Workshop on AOP and Meta-Data for Software Evolution. RAM-SE '09, ACM, Genova, Italy (2009)

34. Gregersen, A.R., Simon, D., Jørgensen, B.N.: Towards a Dynamic-update-enabled JVM. In: Workshop on AOP and Meta-Data for Software Evolution. RAM-SE '09, ACM, Genova, Italy (2009)
35. Gupta, D.: On-line Software Version Change. Ph.D. thesis, Department of Computer Science and Engineering, Indian Institute of Technology, Kanpur, India (November 1994)
36. Gupta, D., Jalote, P.: On Line Software Version Change Using State Transfer Between Processes. Software Practice and Experience 23(9), 949–964 (September 1993)
37. Gupta, D., Jalote, P., Barua, G.: A Formal Framework for On-line Software Version Change. IEEE Transactions on Software Engineering 22(2), 120–131 (February 1996)
38. Herlihy, M.P., Liskov, B.: A Value Transmission Method for Abstract Data Types. ACM Transactions on Programming Languages and Systems (TOPLAS) 4(4), 527–551 (October 1982)
39. Hicks, M., Moore, J.T., Nettles, S.: Dynamic Software Updating. In: ACM SIGPLAN 2001 Conference on Programming Language Design and Implementation. pp. 13–23. PLDI '01, ACM, Snowbird, Utah, United States (May 2001)
40. Hicks, M., Nettles, S.: Dynamic Software Updating. ACM Transactions on Programming Languages and Systems (TOPLAS) 27(6), 1049–1096 (November 2005)
41. Hofmeister, C.R., Purtilo, J.M.: A Framework for Dynamic Reconfiguration of Distributed Programs. Tech. Rep. UMIACS-TR-93-78 (1993)
42. Hofmeister, C.R.: Dynamic Reconfiguration of Distributed Applications. Ph.D. thesis, University of Maryland at College Park, College Park, MD, USA (1993)
43. Kramer, J., Magee, J., Sloman, M., Lister, A.: CONIC: an Integrated Approach to Distributed Computer Control Systems. IEE Proceedings E Computers and Digital Techniques 130(1) (January 1983)
44. Kramer, J., Magee, J.: Dynamic Configuration for Distributed Systems. IEEE Transactions on Software Engineering SE-11(4), 424–436 (April 1985), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=1702024&tag=1,http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=1702024
45. Kramer, J., Magee, J.: The Evolving Philosophers Problem: Dynamic Change Management. IEEE Transactions on Software Engineering 16(11), 1293–1306 (Nov 1990), http://ieeexplore.ieee.org/xpls/abs_all.jsp?arnumber=60317,http://ieeexplore.ieee.org/stamp/stamp.jsp?tp=&arnumber=60317
46. Kuleshov, E.: Using ASM Framework to Implement Common Bytecode Transformation Patterns. Vancouver, Canada (March 2007)
47. Lindholm, T., Yellin, F.: The Java Virtual Machine Specification, Second Edition (1999)
48. Liskov, B., Scheifler, R.: Guardians and Actions: Linguistic Support for Robust, Distributed Programs. ACM Transactions on Programming Languages and Systems (TOPLAS) 5(3), 381–404 (July 1983)
49. Manna, V.P.L.: Dynamic Software Update for Component-based Distributed Systems. In: Proceedings of the 16th international workshop on Component-oriented programming. pp. 1–8. WCOP '11, ACM, New York, NY, USA (2011), http://dl.acm.org/citation.cfm?doid=2000292.2000294
50. Miedes, E., Muñoz-Escoí, F.D.: A Survey about Dynamic Software Updating. Tech. Rep. ITI-SIDI-2012/004, Instituto Universitario Mixto Tecnológico de Informática, Universitat Politècnica de València (2012)
51. Milazzo, M., Pappalardo, G., Tramontana, E., Ursino, G.: Handling Run-time Updates in Distributed Applications. In: 2005 ACM symposium on Applied computing. pp. 1375–1380. SAC '05, ACM, Santa Fe, New Mexico (2005)
52. Murarka, Y., Bellur, U.: Correctness of Request Executions in Online Updates of Concurrent Object Oriented Programs. In: 15th Asia-Pacific Software Engineering Conference (APSEC '08). pp. 93–100. IEEE Computer Society (December 2008)

53. Neamtiu, I., Hicks, M., Stoyle, G., Oriol, M.: Practical Dynamic Software Updating for C. In: ACM SIGPLAN conference on Programming language design and implementation. pp. 72–83. PLDI '06, ACM, Ottawa, Ontario, Canada (2006)
54. ObjectWeb: ASM 4.0 (October 2011), http://asm.ow2.org/
55. Purtilo, J.M.: The POLYLITH Software Bus. ACM Transactions on Programming Languages and Systems 16(1), 151–174 (January 1994)
56. Purtilo, J.M., Hofmeister, C.R.: Dynamic Reconfiguration of Distributed Programs. In: 11th International Conference on Distributed Computing Systems. pp. 560–571 (May 1991)
57. Rellermeyer, J., Alonso, G., Roscoe, T.: R-OSGi: Distributed Applications Through Software Modularization. In: Cerqueira, R., Campbell, R. (eds.) Middleware. Lecture Notes in Computer Science, vol. 4834, pp. 1–20. Springer Berlin, Heidelberg, Newport Beach, CA, USA (2007)
58. Rellermeyer, J.S., Alonso, G.: Concierge: a Service Platform for Resource-constrained Devices. In: Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems. EuroSys '07, vol. 41, pp. 245–258. ACM, Lisbon, Portugal (March 2007)
59. Ritzau, T., Andersson, J.: Dynamic Deployment of Java Applications. In: Java for Embedded Systems Workshop. London, United Kingdom (May 2000)
60. Schmidt, H., Elsholz, J.P., Nikolov, V., Hauck, F.J., Kapitza, R.: OSGi4C: Enabling OSGi for the Cloud. In: Fourth International ICST Conference on COMmunication System softWAre and middlewaRE (COMSWARE '09). COMSWARE '09, ACM, Dublin, Ireland (June 2009)
61. Segal, M.E., Frieder, O.: Dynamic Program Updating in a Distributed Computer System. In: Conference of Software Maintenance. pp. 198–203. Scottsdale, AZ, USA (October 1988)
62. Solarski, M.: Dynamic Upgrade of Distributed Software components. Ph.D. thesis, Fakultät IV (Elektrotechnik und Informatik), Technische Universität Berlin (2004)
63. Solarski, M., Meling, H.: Towards Upgrading Actively Replicated Servers on-the-fly. In: 26th Annual International Computer Software and Applications Conference (COMPSAC 2002). pp. 1038–1043 (2002)
64. Sridhar, N., Pike, S., Weide, B.: Dynamic Module Replacement in Distributed Protocols. In: 23rd International Conference on Distributed Computing Systems. pp. 620–627 (May 2003)
65. Stoyle, G., Hicks, M., Bierman, G., Sewell, P., Neamtiu, I.: Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. ACM Transactions on Programming Languages and Systems (TOPLAS) 29(4) (August 2007)
66. Tatsubori, M., Sasaki, T., Chiba, S., Itano, K.: A Bytecode Translator for Distributed Execution of "Legacy" Java Software. In: 15th European Conference on Object-Oriented Programming (ECOOP '01). pp. 236–255. ECOOP '01, Springer-Verlag (2001)
67. Tavares, A.L.C., Valente, M.T.: A Gentle Introduction to OSGi. SIGSOFT Software Engineering Notes 33(5) (September 2008)
68. Tewksbury, L., Moser, L., Melliar-Smith, P.: Live Upgrades of CORBA Applications Using Object Replication. In: Proceedings of the IEEE International Conference on Software Maintenance (ICSM'01). pp. 488–497. ICSM '01, IEEE Computer Society (2001)
69. Vandewoude, Y., Ebraert, P., Berbers, Y., D'Hondt, T.: Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. IEEE Transactions on Software Engineering 33(12), 856–868 (December 2007)
70. Wang, X., Yang, S., Wang, S., Niu, X., Xu, J.: An Application-Based Adaptive Replica Consistency for Cloud Storage. In: 2010 Ninth International Conference on Grid and Cooperative Computing. pp. 13–17. Nanjing (November 2010)
71. ZeroTurnaround: JRebel 4.5.4 (January 2012), http://zeroturnaround.com/jrebel/