A Survey about Dynamic Software Updating

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-2012/003

A Survey about Dynamic Software Updating

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-2012/003

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

May 7, 2012

1 Introduction

In this report we survey a number of references related to *dynamic software updates* (DSU), which generally speaking can be defined as the application of changes to software that is currently on execution, without having to stop and restart it. This is probably the more general definition of DSU that can be given, at least at a so high level of abstraction. Unfortunately, the different authors that have worked on this topic have not reached a consensus on it and no standard definition of DSU exists yet.

Moreover, a standardized classification of different types of dynamic updates is also missing. Different authors have proposed different classifications, considering their own selection of criteria. For instance, Purtilo *et al.* [59] identify three types of dynamic changes: a) changes to the implementation of a module, b) changes to the structure of the application, this is, changes to the relationships or *bindings* among the modules and c) changes to the *geometry* of the application, this is, to the mapping of that structure onto a distributed architecture (see Section 3.5).

Gupta *et al.* [39, 41] identify three types of updates: a) changes in the structure of the system (for instance, in the relationships among the components), b) changes in the code (for instance, changes in the implementation of existing components) and c) changes in the *geometry* of the system (the assignment of software components to physical nodes, for instance to adapt to a different architecture). See Section 3.7 for additional details.

Buckley *et al.* [23] present a taxonomy of *dynamic update* mechanisms, based on fifteen *dimensions* like the *moment in time* in which they are applied (respect to the development and execution processes), the impact an update on the whole application or the visibility of the history of changes, among others (see Section 3.11).

Giuffrida and Tanenbaum [34] propose a taxonomy of dynamic update mechanisms and also a classification of different types of dynamic updates (see Section 3.23). Panzica [52] proposes a classification of dynamic updates in four classes depending on the type relationship (specialization, improvement, etc.) between the current version of a component and the next version to install (see Section 4.6). Other authors have proposed different classifications, considering different criteria.

In this survey, we are not providing neither a *standard* definition of DSU nor a *classification* of dynamic updates or DSU mechanisms. Instead, we try to keep the survey-oriented nature of the report, by organizing its sections as follows. In Section 2, we show a collection of requirements and goals pointed out as *desirable* according to different authors. This collection is useful to realize about the variety of the criteria considered by the authors and also the variety of contexts in which DSU can be applied. Moreover, it is also useful to realize about the lack of such a standard definition of DSU.

The main part of the survey is showed in Section 3. Then, in Section 4 we present some additional references.

Next, in Section 5 we list a number of concepts, issues and techniques we have identified along all the surveyed references. We conclude the report in Section 6.

2 Requirements and Goals

The collection of requirements and goals that a DSU mechanism must offer varies from one author to another. In this section, the most significant ones are listed.

Fabry [30] identifies the following desirable requirements of a DSU mechanism:

- It should not be necessary to stop the system during a long period of time (but it may suffer a momentary delay).
- The update must not be noticeable by the user, beyond a momentary delay.

Kramer and Magee [48] list a number of *essential* properties related to *configurable software*, as well as some *desirable*, classified in several groups. They identify properties of the programming languages used to build configurable software components, the language used to express both the configuration of the system and the changes to be dynamically applied to that configuration, the underlying operating system itself, the process used to validate the configuration of a system and the *configuration manager* that is in charge to perform the dynamic reconfigurations.

Segal and Frieder [66, 33] identify the following requirements of a DSU mechanism:

- It must allow a new version of a running program to be loaded without having to stop and restart the program. Moreover, the performance should not be affected *too much*.
- It must enable the update of programs written with *current* conventional programming languages and also distributed systems whose size is in the order of several hundreds of nodes.
- It must minimize the amount of user intervention needed.
- It must allow the simultaneous execution of several updates.

Kramer and Magee [49] identify the following requirements:

- *Changes should be specified in terms of the system structure.* Specifically, the changes considered are the addition and removal of nodes and the addition and removal of connection among nodes.
- *Change specifications should be declarative*, instead of *operational*, like those specified by an algorithm.
- *Change specifications should be independent of the algorithms, protocols, and states of the application.*
- *Changes should leave the system in a consistent state.*
- *Changes should minimize the disruption to the application system.*

Purtilo and Hofmeister [59] identify the following requirements:

- *Users need an easy way to configure and invoke a (possibly distributed) application.*
- *Users must have a notation for identifying any of the program components or attributes that they wish to reconfigure.*
- *Users must be able to visualize the current state and geometry of a running program.*
- There must be *a reliable way to coerce the representation of data that is transmitted during both normal communication and any reconfiguration.*

- All communication between processes must be *controlled by the external agent responsible for reconfiguration*. This is to avoid that *uncontrolled* communications interfere in the normal operation of the reconfiguration mechanism.
- *Any reconfiguration mechanism in the execution environment must ensure that all information characterizing a process is captured and represented.*
- *The execution environment needs some way to mark some of the processes as non-relocatable, recognizing that some modules must necessarily act as guards to private resources.*

Sridhar *et al.* [69] identify the following requirements:

- *Initiation*. The update mechanism must be started either by itself or by a third-party (that may be an administrator user, another program, etc.).
- *Module Integrity*. The consistency of the components must be ensured at any time. Tipically, the system controls the interactions among the client code and the components affected by an update, while it takes place.
- *Module Rebinding*. The new version of a component must be loaded and linked in runtime, so new instances of the corresponding objects can be created from then on.
- *State Migration*. When updating an object, its state must be *transferred* to the new version.
- *Instance Rebinding*. The *handles* (like pointers or references) to the old objects must be *redirected* to the corresponding new versions. Moreover, old objects must be *terminated*.

Ajmani, in his thesis [12] (and later, Ajmani *et al.* [13]), identifies the following requirements:

- *Modularity*. The *upgrader* user should not need to know the whole history of updates of a component, but just the current version and the following.
- *Generality*. The update mechanism must allow to apply different types of updates. Two *sub-goals* are considered:
  - *Incompatibility*. *The new version must be allowed to be incompatible with the old one*. The idea is that it may be useful to *relax* de backwards compatibility requirements of the update mechanism to avoid further complications. Keeping the *legacy behavior* is allowed and enforced only when no additional complications are added.
  - *Persistence*. The updates must preserve the persistent state used by the applications (e.g. in databases), although it may be modified.
- *Automatic deployment*. Due to the large size of the target system, updates must be automatable.
- *Controlled deployment*. Updates must be *schedulable* in time.
- *Mixed mode operation*. There may be long periods of time in which there may coexist nodes with older versions of the software and nodes with newer versions.

Solarski, in his thesis [67], identifies the following functional requirements:

- *Basic Deployment Capabilities*. It should be possible to install, enable, disable and uninstall services.
- *Support for distributed services*. It should be possible to install services in the different nodes of a distributed system.
- *Support for co-existence of multiple versions*. The update system should allow to simultaneously run different versions of the same service.

and also some non-functional requirements:

- *Extensibility. A dynamic upgrade support system is required to be easily extensible. It should be possible to add both policies to manage dynamic upgrades in the system and mechanisms supporting these policies.*
- *Portability.* It should be easily portable to other hardware architecture and operating systems.

Moreover, regarding the *algorithm* itself, the following functional requirements are specified:

- *Automated upgrade process.* The update mechanism must be automated as much as possible, and yet it must allow an administrator to manually intervene to solve problems if some failure happens.
- *System consistency preserved during the upgrade.* The update mechanism should not lead the system to an *inconsistent* state, although the exact definition of *consistency* depends on the application itself.

and also these non-functional requirements:

- *Minimizing the loss of the system functionality during the upgrade process.* In case the dynamic update process degrades some part of the system, this must be small and well isolated.
- *Minimizing the unavailability periods.* The update process should interrupt the application the shortest time possible.
- *Dependability of the upgrade.* The update process must be *atomic*. In case some failure happens, a *rollback* procedure should be performed.
- *Upgrade transparency.* The update process must be *invisible* for those parts of the system that are not directly related with it.

Some other requirements, related with the management and coordination of the updates are also identified:

- *Automated upgrade management.* There should be some mechanisms to decide in which moment an update has to be applied. Typically, a number of alternatives can be considered: just after processing an update request, when the component to update is found *idle* or at any other point in time.
- *Support for multiple simultaneous upgrades.* It should be possible to apply different updates simultaneously.

Finally, some requirements of the components to update are also specified:

- *Orthogonal upgradability.* The part of the components that is *updateable* must be clearly away from the rest of the funcionality.
- *Simplicity of development of upgradable components.* The application programmer should not make a big effort to create the *dynamically updateable* part of the components. Preferably, this part should be provided by the update mechanism itself, in a more or less automatic way.
- *Minimizing the set of constraints on the system.* The dynamic update mechanism should impose the smallest number of constraints to the user applications and the whole system in general.
- *Heterogeneous service support.* The update mechanism should allow the update of *heterogeneous* components (for instance, using different technologies, models, programming languages, etc. even among different versions of the same component).

Hicks and Nettles [44] identify the following requirements:

- *Flexibility.* Any part of a running system should be updateable without requiring downtime.
- *Robustness.* A system should minimize the risk of errors and crashes due to an update, using automated means to promote update correctness.

- *Ease of use*. Generally speaking, the less complicated the updating process is, the less error-prone it will tend to be. The updating system should therefore be easy to use.
- *Low overhead*. Making a program updateable should impact its performance as little as possible.

Chen *et al.* [24] identify the following requeriments:

- *Binary Compatibility*. The dynamic update mechanisms must offer *backward compatibility*, i. e., they must allow the update of existing programs (already compiled, in binary form) and even the update of currently running programs.
- *Multithreading Support*. The update mechanisms should be able to update multithreading programs.
- *Recovery of Tainted Stated*. The update mechanism should be able to restore a correct state of an updateable program from a *tainted* or *corrupted* state.
- *Usability and Manageability*. The update mechanism must be easy to use and offer to the user operator mechanisms to control the update processes (for instance, the roll-back of updates, even of those already confirmed).
- *Low Overhead*. The update mechanism should impose the lowest overhead possible to the updateable program.

Murarka and Bellur [54] identify two general requirements:

- *Correctness*. A dynamic update should not lead the system to an incorrect state.
- *Continuity*. During and after a dynamic update, the system must go on offering its service, suffering the smallest interference possible.

Gregersen and Jørgersen [36] list a number of requirements, *extracted* from other papers:

- *Programmer transparency*. The application programmer should not need to know too many details about the update mechanism and this should not modify significantly their workflow.
- *Flexibility*. The update mechanism should allow to apply many different types of updates (ideally, it should allow to apply any update that could be made under a *stop, redeploy and restart* operation mode).
- *Performance*. The performance overhead imposed to the application should be minimized.
- *Correctness*. The behavior of the application must be the same than the one that may be 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.
- *Concurrency*. The update mechanism should allow the update of multithreading applications. It should no provoke any deadlocks or any other related issue.
- *Availability*. The dynamic updates should not reduce the application availability.
- *Configurability*. The update mechanism should admit a number of *update policies* to configure and tune its operation.
- *Roll-back*. The update mechanism should allow to *roll-back* the updates (although it will not always be possible).

Bannò *et al.* [17] identify the following goals:

- Guarantee the data consistency.
- Guarantee the *consistency of flow*. Before updating a component, the update mechanism must *wait* until it finishes processing all the pending requests.

- Guarantee the *semantic consistenciy* in the *sequences* of invocations that are interrupted by an update.
- Avoid blocking too many parts of the application, to minimize a significant performance impact.
- Guarantee that the update of several components is *atomic*.

Panzica [52] proposes an update mechanism that satisfies the following goals:

- *Granularity*. The update is performed at an architecture level, considering the relationships among the components.
- *Locality*. The update considers information local to the component to update.
- *Correctness*. The update allows the system to behave normaly, including a normal termination of the on-going requests.
- *Timeliness*. The update is carried in a *short period of time*.
- *Disruption*. The update interrupts the normal operation of the application during the shortest time possible.
- *Human effort*. The update is automated and the user intervention is minimized.

3 Main Results

In this section, a number of *main works* are listed. These works are considered important because they usually propose a new procedure or technique and they are usually referenced by many other subsequent works.

3.1 *How to Design a System in which Modules can be Changed on the fly*, Fabry, 1976

Fabry [30] is one of the first authors to identify the need to dynamically update parts of a software system. He also identifies different levels of *complexity* of dynamic update mechanisms, depending on whether or not the application has some state that should be persistent.

Moreover, he proposes a basic dynamic update mechanism based on two key techniques: *adding a level of indirection* (see Section 5.3) and *rewriting* some parts of the application at a binary level (see Section 5.2). The mechanism is illustrated with and example in which a function is updated.

First, a level of indirection is added between the updateable function and a call to it from the client code. This indirection level consists of a low level JMP-like instruction, to *jump* to the definition of the function. When the function is updated, a new version of the function is installed and the jump instruction is modified (directly editing the low level binary instruction) to make it point to the new definition of the function.

The paper includes two more examples, also based on this idea, which cover two other situations.

3.2 *Dynamic Module Replacement in a Distributed Programming System*, Bloom, 1983

Bloom's Ph. D. thesis [21] tackles the problem of dynamic update of distributed systems in the context of the Argus programming system [51]. His main goals are to clearly define the semantics of the process of dynamically updating a distributed system and under which conditions it can be performed, in a *safe* manner. Moreover, he introduces a mechanism developed to support the dynamic update of Argus programs. The mechanism depends heavily upon the Argus libraries and runtime, but the basic idea behind the mechanism is to use some redirection technique (see Section 5.3 for other forms of *indirection*) to *remap* some *handlers*. Before applying the update, a state transfer must be performed, to preserve the state used by the part of the program to be replaced (see Section 5.5).

That work is one of the first to identify several issues related with the dynamic update (especially that of distributed systems). For instance, it identifies the *syntactic* gap among a data type that has been updated from version $v1$ to version $v2$ and the data used by that type, created or modified by its version $v1$. He is also one of the first authors to identify the need of using *transformation functions* (see Section 5.6) to *migrate* the data accordingly to the updates of the data type that handle them.

The thesis takes some ideas from [30] like the use of version numbers to identify the different versions of the data types and the data items themselves. It uses them to perform the necessary checks in runtime and if necessary, invoke the proper transformation functions.

3.3 *Dynamic Program Updating in a Distributed Computer System*, Segal and Frieder, 1988

Segal and Frieder [66, 33] define a mechanism for dynamically updating procedures in a distributed system. This mechanism should be transparent from different points of view: a) the source code should no need to depend on the update mechanism and b) the need of human intervention should be minimized. Moreover, it should impose the least performance overhead possible.

The authors define the concept of *active* and *inactive* procedures, used to restrict the dynamic update mechanism they propose.

First, a procedure is *active* if it is in the runtime stack (this is, if it is being executed) or if its *new version* is able to call any *active* procedure. Otherwise, the procedure is *inactive*.

To update a procedure P, the following constraints must be fulfilled:

- P must be inactive
- All the procedures *reachable* from P are also inactive

In a certain sense, such a requirement can be seen as a certain form of *quiescence* like the one originally defined in [49] (see also Section 5.1).

Moreover, the authors also define the concept of *semantic dependency* of a procedure P like the set of procedures P depends on. This dependency does not need to be *syntactic* but can be *semantic* (in this case, it must explicitly be declared by the programmer, thus reducing the transparency of the solution).

Given these two definitions, the authors propose a mechanism to dynamically update a procedure P and the set of all the procedures that semantically depend on P.

The mechanism relies on additional mechanisms and techniques. First, it uses *mapper procedures* which are procedures that are written by the programmer to transform data structures from an old version to a new one. These procedures are in essence similar to the *transformation functions* used by other authors (see Section 5.6). Furthermore, it uses *interprocedures* which are intermediary procedures used to *redirect* the invocations to the *old versions* of the procedures to the *new versions*. If needed, they are even able to hide to the old version of a procedure, the existence of new versions of the data types so it can go on operating with the *old data types* it knows. The second type of *indirection* used by the authors is based on a *binding table* which holds *pointers* to the updateable procedures. These pointers are overwritten in runtime, 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. See Section 5.3 for other forms of *indirection*.

3.4 *The Evolving Philosophers Problem: Dynamic Change Management*, Kramer and Magee, 1990

In [49], Kramer and Magee (1990) present one of the seminal works about dynamic software update. They propose a mechanism to manage *changes* in a distributed system. The types of changes considered are the addition or removal of nodes and the addition or removal of links between nodes. The goals of such a mechanism are listed in Section 2.

The idea of *consistency* used in those goals refers to the state of the communications among the nodes. A system is in a *consistent state* if none of its nodes has pending transactions (seen as *data exchanges*). As some types of changes can violate the *consistency* of a system (for instance, removing a node or a link), the update procedure must be careful and avoid applying changes that may violate the consistency of the current system.

In [49], the authors present a number of relevant concepts and ideas, used later, directly or indirectly, by many other authors. One of these key ideas is the concept of *quiescence* (see Section 5.1) which is related with the concept of *activity* of a node (see its Section III.D). A regular node is usally *active* in the sense that it can be sending and receiving data to and from other nodes. On the contrary, a *passive node* is a node which a) is not currently taking part in any transaction started by itself and b) is not going to start a new transaction. The *passive set* of a node is a set composed of a) the node itself and b) the set of nodes that may start data exchanges with it. It's worth noting that such a set can be *sintacticaly deduced* if the system uses a declarative way to specify the relationships among the nodes like the one proposed by the authors in [47, 48].

A *quiescent node* is a node such that *all the nodes in its passive set are passive*. Informally, this means that the node is not involved in any communication with other nodes, this is, the node is *quiet* and moreover, all the nodes that can communicate with it are also *quiet*. In Section II.E, the authors reason about the *reachability* of the passive state (considering that any transaction or data exchange finishes in a bounded time).

Based on these concepts, the authors propose the following procedure to dynamically update a set of nodes:

1. Calculate the set of nodes that must be *quiescent*
2. *Passivate* those nodes (this is, ensure they enter in a *quiescent* state)
3. Apply the change (unlink, remove, create and link nodes)
4. Reactivate the passivated nodes

The complete procedure is described in Section IV.A of [49].

For this procedure to work, the application must collaborate. 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). Moreover, 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.

This kind of connections between the update mechanisms and the managed processes or applications has been also found in some other papers (see Section 5.4). In fact, in [49], the authors argue about the need of defining some kind of *standard interface* to communicate the update mechanisms and the applications. Unfortunatelly, these requeriments are opposite to the *transparency* requirements and goals usually requested by some of those papers (see Section 2).

As shown in Section 5.1, the concept of *quiescence* has been directly or indirectly used by many other authors to express some *stability* requirement. Nevertheless, it also has been criticized because it may impose a significant blocking. For instance, in a distributed system in which all the nodes are strongly coupled, the whole system may become completely (temporarily) *stalled*. To avoid this problem, in Section VII of [49], the authors suggest to apply a relaxed definition of *active* and *passive* nodes. The idea is that a node could be active respect to a given connection and passive respect to another. They also suggest that nodes may be grouped in order to restrict the scope of the *passivization* to the nodes of a group and thus avoid blocking the whole system. Regarding this subject, in [74] Vandewoude *et al.* (2007) argue that the *quiescence* concept proposed by [49] is too strict and propose a relaxed form of *quiescence*, called *tranquility*.

3.5 *Dynamic Reconfiguration of Distributed Programs*, Purtilo and Hofmeister, 1991

Purtilo *et al.* [59, 58] present POLYLITH, a software bus to build distributed systems that can be dynamically *reconfigured*. The idea is that the programmer also specifies some specification of the program using its own configuration language. These allows POLYLITH to know about the modules of the program, their interfaces and some other details. During the compilation phase, POLYLITH also *compiles* this specification and produces some proxies that intercept the invocations to the functions of the modules (see Section 5.3 for other forms of *indirection*).

The basic idea behind the reconfiguration mechanism used by POLYLITH consists in using an *abstract* format to characterize the state of the modules, as proposed in [42] (see Section 5.5 for other references related to state transfer). This allows to get the state of a running module before updating it and then restore it back or even move a module from a physical node that uses a given architecture to a different node that may use a different architecture.

Regarding the update procedure, it is performed by the bus and the proxies. The considered updates are the change of the implementation or interface of a module or the change of the *bindings* among modules (through their interfaces). The basic idea is that to perform a dynamic reconfiguration, one or more *capabilities* must be obtained. Once got, the updates are applied and finally atomically committed.

3.6 *On Line Software Version Change Using State Transfer between Processes*, Gupta and Jalote, 1993

Gupta *et al.* [40] propose a procedure to dynamically update a process. Roughly speaking, the procedure consists of the following steps:

1. Start a new process with the new version of the binary code
2. At some point, transfer to the new process the state of the original process
3. Transfer the control to the new process

They also provide an implementation of the procedure (for programs written in C, running in Sun 3/60 machines under SunOS). The implementation includes an overwrite of some system calls like `open` and `close`.

The mechanism tries to apply the updates when it is safe. To apply an update that consists in modifying a given procedure `p` of the process, the update mechanism tries to ensure that `p` is not currently on execution by looking for it in the stack of the process. If the procedure is not in the stack, then it is safe to apply the update. This check tries to ensure some *stable* state, which is, in a certain sense, some sort of *quiescent* state (see Section 5.1 for other forms of *quiescence*). See also Section 5.5 for details about other references that use some state transfer technique.

3.7 *On-line Software Version Change*, Gupta, 1994

The Ph. D. thesis by Gupta [39] presents a formal framework to model changes to be dynamically applied to running software and reason about the validity of such dynamic changes, considering that a dynamic change is *valid* ”if some time after the change, the process reaches a reachable state of the new program version” (see also [41]).

He considers software systems of several types: non-structured imperative, structured imperative, object-oriented and distributed. For each of them, he proposes sufficient conditions that must be fulfilled in order to ensure that the dynamic changes are valid.

The thesis includes a prototype implementation of a dynamic update mechanism for programs written in C and a performance study that shows that the dynamic update mechanism causes a little disruption.

3.8 *Towards Upgrading Actively Replicated Servers on-the-fly*, Solarski and Meling, 2002

Solarski and Meling [68] propose a procedure to dynamically update a distributed system that uses *active replication*.

The following assumptions are made:

- The server is updated atomically. Two update processes cannot interleave.

- While a replica is being updated, it cannot process client requests.
- *Input conformance.* The new version of a replica accepts the input acceptable by the old version.
- There exists a *mapping* from the state of the old version of a replica to a next version.
- *Output conformance.* The output produced by the new version is the same that the previous version would produce (if the input provided is acceptable to that previous version)[1].
- *Upgrade atomicity with respect to client upgrades.* Once an update from an old version to a new version is performed, clients only provide input acceptable to the new version.

To update a distributed system composed by a set of replicas, first a *set of candidate replicas* is defined, initially containing all the available replicas. Then a number of steps are followed:

1. Choose a *candidate* replica from the set of candidate replicas (observing the assumptions pointed out above: for instance, it can't be serving requests, etc).
2. Check that it can be updated.
3. If so, *shutdown* the replica, apply the software update, restart it, update its state from the states of other replicas and remove it from the set of candidate replicas.
4. If not, choose another replica and proceed.

These steps are repeated for all the available replicas, until all of them get updated.

It must be noted that the procedure used to update the replicas may be a regular one, since it is applied when they are stopped.

On the other hand, according to the assumptions pointed out above, we can assume that some kind of *state transfer process* (see Section 5.5) is performed when a replica is updated.

3.9 *Dynamic Module Replacement in Distributed Protocols*, Sridhar *et al.*, 2003

Sridhar *et al.* [69] present a technique for replacing software modules, which is based on the use of intermediary objects (see Section 5.3). 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#).

3.10 *Dynamic Upgrade of Distributed Software Components*, Solarski, 2004

The Ph. D. thesis by Solarski [67] includes the proposal of three dynamic update algorithms, for centralized systems and distributed systems that use the *active replication* and *passive replication* (see Section 5.11).

The first algorithm proposed is addressed to *centralized* systems. It is very basic and limited but can be used as a reference in comparisons with the next algorithms. Basically, this algorithm is composed by the following steps:

1. Install the new version of the component
2. Deactivate the old component, transfer the state, rebind the connections
3. Activate the new version of the component
4. Uninstall the old version of the component

[1]Our main concern about this assumption is that it prevents some bugs to be fixed. For instance, it may happen that a given version of an application produces some output for a given input and the output is wrong due to some bug in the code. In this case, should the next version of the application be forced to produce the same wrong output for the given input? Instead, the output should be allowed to change, according to the proper fix of the bug.

As the authors say, the component is yield unavailable during the second step (which means that the service is interrupted). Moreover, although it is not explicitly said, it assumes that two versions of the same component can, at least, be installed simultaneously.

The second algorithm addresses the dynamic update in distributed systems that use the *active replication* technique. It was previously presented in [68]. A brief sketch can be found in Section 3.8.

The third algorithm addresses the dynamic update in distributed systems that use the *passive replication* technique. It has the following steps:

1. Create a new replica, with the new version of the software and add it as a passive replica, to the set of secondary replicas.
2. Update, one by one, all the original secondary replicas. For each one:
   (a) Create a new updated replica
   (b) Add it as a passive replica, to the set of secondary replicas
   (c) Stop the original secondary replica
3. Force a *failover* so one of the passive replicas is promoted to primary
4. Stop the original primary replica

Although in some cases it is not explicitly said in the procedure sketches provided in [67], the three of them depend on some *state transfer* step to *copy* the state of a current component to the new version of the component (see Sections 3.4.1.3 and 4.1 of [67]). The *state transfer* topic is covered in Section 5.5.

3.11 *Towards a Taxonomy of Software Change*, Buckley *et al.*, 2005

Buckley *et al.* [23] present a taxonomy of *dynamic update* mechanisms, based on fifteen *dimensions*, grouped in four *themes*:

- Temporal properties: Time of change, Change history, Change frequency, Anticipation
- Object of change: Artifact, Granularity, Impact, Change propagation
- System properties: Availability, Activeness, Openness, Safety
- Change support: Degree of automation, Degree of formality, Change type

3.11.1 Description of the *dimensions*

Time of change. It can be *static* (changes are applied at a source code level), *load time* (changes are applied at load- or link-time) and *dynamic* (changes are applied in runtime).

Change history. The *change history* is the set of changes applied to a given system or application. The update mechanism *may* or *may not* render visible and publicly available the sequence or history of changes:

- The history of changes is visible. This case is covered by *version control* tools.
  1. The versioning of the changes may or may not be supported:
     - The versioning is *static*. In runtime, there is a single version of each component.
     - The versioning is *dynamic*. In runtime, different versions of a component can coexist.
  2. The changes may be applied in different ways:
     - The changes are applied *sequentially*: changes from different users are allowed but not at the same time.
     - The changes are applied *in parallel*: changes from different users can be applied concurrently.

        - The changes are *synchronous*: all the users apply their changes to the same copy of the data.
        - The changes are *asynchronous*: each user may apply the changes to a different *copy* of the data.
            - The changes are *convergent*: the changes from all the users are merged into a single copy.
            - The changes are *diverging*: each user keeps a different copy of the data.

- The history of changes is not visible. The changes are applied destructively.

Change frequency. The changes may be applied *continuously* (when there is something to change), *periodically*, or *at arbitrary intervals*.

Anticipation. The changes can be *anticipated* (when they are *foreseen during the initial developement of the system*) or *unanticipated* (they typically arise during the deployment or exploitation of the system).

Artifacts. The changes can be applied to the *architecture*, the *design*, *source code*, *documentation*, *configurations*, *test suites* or a combination of them.

Granularity. The *granularity* of the changes can vary. For instance, changes to the architecture of a system can be *coarse* if they affect the whole system or *fine* if they affect just a subsystem.

Impact. The *impact* of the changes can also vary, from *local* (when a change affects a small part of the system) to *global* (when it affects to the whole system).

Change propagation. Once applied a change, it may be necessary to propagate it to some other parts of the system. The authors point out the use of *completely automated propagation mechanisms* in contrast to a *completely manual change prograpagion*. Between these two alternatives, we can identify a third type of change propagation that could be called *user-assisted change propagation*. Refactoring tools like those used in current development suites like Eclipse or Visual Studio typically offer user-assisted change propagation mechanisms but at the same time, they offer the possibility to completely automate the process.

Availability. An update mechanism can be classified according to the availability needs of the system to update. Some mechanisms force the system to be paused or even halted and restarted (thus reducing its availability) while others allow the system to remain available.

Activeness. From the point of view of *how the change is started*, systems can be *reactive* if the change has to be started *externally* (for instance, by a user) or *proactive* if the system itself decides to start the change (which is typical in self-adaptive systems, self-healing systems, auto-reparing systems, etc.).

Openness. The update mechanism may be:

1. *Open*, if it is targeted to systems designed to be extended (for instance, by means of plugins).
2. *Partially open*, if it is not open but still offers some limited capacities to be extended.
3. *Closed*, if the target system is not updateable (beyond the update of its source code).

Safety. The update systems can offer different kinds of *safety* mechanisms:

- *Security*, as a synonym of *protection of the software*. Typically includes mechanisms to protect the software from malware, unauthorized users, etc.
- *Behavioral safety*, this is, a *correct behavior* of the software regarding to its specification.
- *Backward compatibility* (when a change is applied to a system, it can go on working as before the change).

Degree of automation. The *degree of automation* can be *automated*, *partially automated* or *manual*.

Degree of formality. The *degree of formality* used to express the changes can vary from a *very formal level* (for instance, like in [47, 48]) to a completely *manual* (for instance, by applying changes to the source code).

Change type. The changes can be:

- *Structural*, if they affect the structure of the system, like the *addition* of new parts of the software, the *subtraction* or the *alteration* of existing parts of the software. Another classification can be found in [52] (see Section 4.6).
- *Semantic*, if they affect the behavior of the system. They can be *semantic-modifying* or *semantic-preserving*, depending on whether they modify the semantics of the system or preserve it.

3.11.2 Application of the *dimensions*

The authors apply this taxonomy to three systems related to software changes:

1. Refactoring Browser [64], a browser for Smalltalk IDEs (like VisualWorks, VisualWorks/ENVY and IBM Smalltalk).
2. CVS [2]
3. eLiza [57], a technology *"to create self-managing servers and networks in four ways: self-configuring, self-healing, self-optimizing and self-protecting"*.

3.12 *Dynamic Software updating*, Hicks and Nettles, 2005

Hicks and Nettles [44] present a framework that allows the dynamic update of running programs written in a C-like programming language. The authors ensure that it is the first framework that ensures the *type-safeness* of the updated systems. The update mechanism uses *dynamic patches* that consist of *verifiable native code* (regular binary code that includes *"annotations that allow online verification of the code's safety"*).

The proposal allows the programmer to update the code of the software, the definition of the data types used and the data it handles. Moreover, the programmer can also decide when an update has to take place and also mark some parts of the code that should not be interrupted by a dynamic update (see Section 5.4).

The framework is based on dynamic patching and applies different techniques, also found in other papers.

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 5.3), 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 (see Section 5.2). 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 (this technique is also used by other authors, see Sections 5.2 and 5.7). 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 5.6) 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.

3.13 *Handling Run-time Updates in Distributed Applications*, Milazzo *et al.*, 2005

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 5.3).

This layer includes a new Java class loader that uses some Java rewriting techniques (see Section 5.2) 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.

3.14 *Modular Software Upgrades for Distributed Systems*, Ajmani *et al.*, 2006

Ajmani *et al.* [13] describe a methodology to dynamically update objects of a distributed application. Their methodology allows an *updater user* to send, during the regular execution of the application, a number of *update requests*. These consists of changes to the definition of the object types. Some of the changes are *compatible* (they can be directly applied, without disturbing the normal operation of the application, like adding methods or changing their implementation). Others are considered *incompatible* (they can pose problems to the application, like removing methods or changing their signature). The proposal allows both types of changes.

The proposed methodology includes the use of some special objects called *simulation objects* (see Section 5.3) to represent past and *future* versions of the current objects. These simulation objects allow the nodes of the distributed application to be updated in differents points in time. Each node can access a different *version* of an object, just by transparently accessing a different simulation object. As a result, there's no need to update all the nodes simultaneously.

The proposal also allows the user to define *scheduling options* (see Section 5.12), to specify how to schedule the update of the nodes (there are many alternatives: all the nodes simultaneously, half of the nodes first and then, the rest, etc.). The user can also define *transform functions* (see Section 5.6) to specify how to update the type of an object, from its current version to the next one. These functions are especially important to decide how to perform the *incompatible* updates.

3.15 *POLUS: A POwerful Live Updating System*, Chen *et al.*, 2007

Chen *et al.* [24] 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 5.7 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 5.3 for other forms of level indirection).

Moreover, POLUS is distinguished by the possibility to *reverse* this procedure (also see Section 5.13). 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 5.13 other examples of rollback-enabled mechanisms are given.

Another significant feature of POLUS is that it can be used to update multithreaded programs.

An extended version of this paper can be found in [25]. It includes an extended description of some experiments made with POLUS to dynamically update three different systems: a FTP daemon (vsftp), a SSH daemon (OpenSSH) and a web server (Apache httpd).

3.16 *Mutatis Mutandis: Safe and Predictable Dynamic Software Updating*, Stoyle *et al.*, 2007

Stoyle *et al.* [70] present Proteus, a formal system to model and apply dynamic updates in imperative single-threaded programs, while ensuring a *conn-freeness* (type correctness) property. This property tries to ensure that once some code is updated to a new version, the data used with the new version must be no longer used with a previous version.

This system includes its own programming language and compiler and runtime, among other tools and resources.

The basic idea consists in statically analyzing the source code of the program to update and identify points in which the updates can be applied while preserving the *conn-freeness* property. For each point, it identifies which data types are not *conn-free*, this is, which ones should not be updated in order to keep the *conn-freeness* property. In run-time, Proteus performs the necessary checks and ensures that none of those types are updated.

See Section 5.7 for other references that use some sort of (static) analysis of the source code.

3.17 *R-OSGi: Distributed Applications Through Software Modularization*, Rellermeyer *et al.*, 2007

Rellermeyer *et al.* [60] propose R-OSGi, an extension of OSGi [7, 14] that works with distributed systems. A standard implementation of OSGi allows to build a Java application from a number of *bundles*, that run in the same Java Virtual Machine. With R-OSGi, each bundle can run in a different Java Virtual Machine. Moreover, R-OSGi hides the distributed nature of the bundles, which means that the bundles do not need to know if the other bundles are local or remote. This transparency level is more or less similar than the one provided by other middleware platforms like RMI or CORBA. In addition, R-OSGi is even able to hide some remote errors and *encapsulate* them in form of regular OSGi *disconnection events*, so the local bundle can go on being strictly OSGi compliant.

In [60], they present a comparison among R-OSGi, RMI and UPnP and show that R-OSGi can yield better performance numbers.

See Section 5.9 for additional references about OSGi.

3.18 *Consistently Applying Updates to Compositions of Distributed OSGi Modules*, Rellermeyer *et al.*, 2008

Rellermeyer *et al.* [62] consider the problem of guaranteeing the *consistency* among the bundles of an R-OSGi application (see Section 3.17).

The problem is tackled from a *syntactic* point of view and can be summarized as follows. In a regular OSGi (and R-OSGi) application each bundle has a metadata manifest file in which it *declares* some *static*, syntactic relationships: which packages it exports and which packages it imports from other bundles (this is, packages it depends upon). Thus, when a bundle A imports a package from a bundle B, then there is a dependency of A upon B. The problem appears when B has to be updated, since there are some types of updates that may cause *inconsistencies* (for instance, just uninstalling B).

The solution proposed in [62] consists in offering some sort of *deferred updates*. For instance, if the bundle B has to be uninstalled, R-OSGi first checks if some other bundle depends on B. In such a case, the operation is considered *not safe*. Then, the bundle is kept but only for those bundles that depend on it. For the rest of the bundles, it *appears* as uninstalled.

If the bundle is not uninstalled but just updated, a similar solution is applied. The old version of the bundle is kept for those bundles that declare a dependency on it while it appears as updated for the rest of the bundles.

The pending (or *deferred*) changes made to the bundle B (its uninstallation or update) are finally made effective when R-OSGi detects that no other bundles can use it any more. This happens when the bundles that declared dependencies upon it are finally uninstalled or updated. In the later case, their new versions automatically *see* the new version of B.

R-OSGi also offers the service *PackageAdmin* which is an official but optional part of OSGi. This service can be used to force a *package refresh*, which forces one or several bundles to be reloaded. As a result, a reloaded bundle *forgets* its old dependencies and starts seeing the new versions of their imported bundles.

See Section 5.9 for additional references about OSGi.

3.19 *Correctness of Request Executions in Online Updates of Concurrent Object Oriented Programs*, Murarka and Bellur, 2008

Murarka and Bellur [54] study how to correctly apply dynamic updates to multithreaded object-oriented programs. Given an application composed of a number of objects and threads that communicate by sharing data among the objects, the goal is to be able to apply run-time updates to the application so two major properties are guaranteed: *correctnetss* (the program behaves correctly during and after the update) and *continuity* (the program goes on providing its service during the update).

Due to the relationships among the threads it is not possible to apply any update to any class at any moment. For instance, there may be ongoing invocations that must be left to finish (using the *current* class definitions) while other requests might be executed with a *new* definition of the classes.

To know if an update to a class can be *correctly* applied, it is necessary to analyze the impact that change may have upon the objects of that class. On one hand, there may be some changes in the source code that may not change the state of the objects, like adding a new method to a class. On the other hand, there are other types of changes that will cause changes in the state of the corresponding objects, like changing the type of an attribute of a class.

Thus, when applying a change to update an *old version* to a *new version*, a) there may be objects created with the old version that cannot be used with the new version of the class and viceversa, this is, b) there may be objects created with the new version that cannot be used with the old version of the class.

Given an update of a class B from a version $v1$ to a version $v2$, then two types of *compatibility* can be defined. First, B is *backward state compatible* if the objects created with version $v1$ of B can be used with version $v2$. Classes that are not *backward state compatible* must be *transformed* by means of user-provided transformation functions (see Section 5.6). On the other hand, B is *forward state compatible* if the objects created with version $v2$ of B can be used with version $v1$.

The authors then define the following *Request Execution Criteria*[2], to decide if an invocation of a method of a class that is being concurrently updated must use either the *old* or the *new* definition of the class.

- The *Old Program Execution* criterion is defined as follows: a request must use the *old* version of a class if both constraints are fulfilled: a) the request does not access to any object with an *old* version that cannot be used with the new version *after* the update is applied (in that case, it should use the new version) and b) the request does not access to any object with a *new* version that cannot be used with the old version *after* a new request has accessed the object (it should use the new version).
- The *New Program Execution* criterion is defined as follows: a request must use the *new* version of a class if the request does not access any object with an old version that cannot be used with the new version (in that case, it should use the old version).

[2]The idea behind these criteria is very similar to that of the *conn-freeness* property in [70].

To check if these criteria are met, the proposal of the authors consists in analyzing the source code of the classes of a program and building a graph that relate them, according to their use of shared objects. They use some graph theory procedures and as a result, they identify points in the source code of the classes to update, called *update points* in which it is possible to apply the update in a *safe* manner (without deadlocks and guaranteing correcteness and continuity).

In Section 5.7 we surveyed some other references that use some sort of static analysis (for instance, of the source code).

3.20 *Dynamic Update of Java Applications—Balancing Change Flexibility vs Programming Transparency*, Gregersen and Jørgensen, 2009

Gregersen and Jørgensen [36] 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 [36] 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 5.3 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 (see Section 5.2).

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 5.5). 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 5.13 for other authors that also offer some sort of *rollback* mechanism).

3.21 *OSGi4C: Enabling OSGi for the Cloud*, Schmidt *et al.*, 2009

Schmidt *et al.* present *OSGi for the Cloud* (OSGi*4C*) [65], an OSGi implementation that offers some improvements respect to the standard specification.

In particular, it is an extension of the regular OSGi specification, designed to offer the standard dynamic update features to distributed and cloud systems. Basically, the idea is to offer the possibility to build up distributed systems from OSGi bundles that are *physically distributed*. OSGi*4C* allows an application to *reference* and download remote OSGi bundles.

It also allows the re-deploy of just a part of a node. When a node of a distributed or, specifically, a cloud system has to be updated, instead of updating the whole node (for instance, by deploying a new version of the whole *node image*, which could be extremely large), OSGi*4C* allows to specify and apply some kind of *patch* which should be, generally speaking, much smaller than the whole node image.

OSGi*4C* is built on top of JXTA [35]. It is claimed to be a better solution than any others like Java Web Start (which is actually nor a cloud neither a *dynamic update* solution), the standard OSGi Bundle Repository service, and other non-standard solutions like SATIN [76]. Moreover, it is *compatible* with other solutions like R-OSGi (see Section 3.17 and [60]).

In Section 5.9, other references related to OSGi are surveyed.

3.22 *Handling Consistent Dynamic Updates on Distributed Systems*, Bannò *et al.*, 2010

Bannò *et al.* [17] analyze the problem of keeping the consistency during the dynamic update of a distributed system.

First, they identify three *levels* of consistency that must be kept when updating a component from a given version to the next one:

- Consistency of data. The data available to the first version must be somehow *transformed* and *transferred* to the next version (see Section 5.5).
- Consistency of flow. The update must not stop the *flow* of ongoing requests. If it gets interrupted to apply the update, it must be resumed as if the change never had happened. This can be achieved leading the system to some kind of *quiescent* status (see Section 5.1).
- Semantic consistency. There may be semantic relationships between different parts of the application or system that must be preserved when applying a dynamic update. A typical example is a transaction that involves computation in different components. If a dynamic update is performed during the executing of a transaction, it should ensure that the result obtained is *semantically consistent*.

They argue that to keep the semantic consistency and guarantee the validity of the update procedure, there must be some *coupling* between the application and the update mechanism. This coupling is actually twofold. First, the application must be designed in some special way to consider dynamic changes. The programmer must also need to provide some meta-information regarding each update to apply, including a specification of which parts of the application may be interrupted by an update and which ones should not. Second, in run-time, the application must offer some support to the update mechanism. Some other authors also identify the need of such an *intrusive coupling* between the application and the update mechanism (see Section 5.4).

In [17] they present the FREJA framework that supports transparent dynamic updates of distributed systems written in the Java language. This framework is based on the use of specific class loaders, some (centralized) update *managers* and some intermediary objects that control de execution of updateable components (see Section 5.3). They also outline the key points of the procedure they use to update a component. The first step is to lead the system to a some kind of *quiescent* state (see Section 5.1). This includes computing the set of classes and methods that could invoke the component to replace (which is actually some form of *passive set* as in [49]) and wait until the current *executions* end. Moreover, the start of new executions is temporarily paused. The next step, consists in aplying the update, by means of code rewriting techniques (see Section 5.2). Finally, the executing of the blocked components is resumed.

3.23 *A Taxonomy of Live Updates*, Giuffrida and Tanenbaum, 2010

Giuffrida and Tanenbaum [34] study the topic of dynamic software updates. Specifically they focus on the updates of operating systems.

In their study they propose a taxonomy of *live updates focusing on the nature of the update*. The taxonomy is based on a rough classification of dynamic changes in the following set of classes, of increasing *complexity*:

1. Changes to code. They are changes that only affect to the implementation of the program or services.
2. Changes to data. They can affect to both volatile and persistent data.
3. *Resource-sensitive changes*. They affect to basic resources (typically hardware, like main memory, disk, etc.).

They also define the concept of *structural unit* as the minimal unit that is updateable. It could be a function, an object or a whole process. The concept is used to define some of the classes of the proposed taxonomy:

- Changes that affect to a single structure.

- Changes (to code or data) that affect to the protocol among two or more structural units.
- Changes that affect to global data structures, shared among several structural units.
- Changes in global algorithms that affect to several structural units.
- Changes that affect to persistent data (for instance, data stored in disks).
- Changes that affect to the hardware resources (for instance, changes in the minimal hardware requirements).

Moreover, the authors draw a number of conclusions. For instance, they conclude that dynamic updates are not always feasible. In the context of operating systems, it may happen that the update is so complex that a full reboot or the manual intervention of a human administrator is needed. By contrast, sometimes the dynamic update although possible, is not desirable (for instance, when it causes a significant *disruption* to the system).

The authors conclude by defending that the best approach to build updateable software is to design it so it is aware of its *updateable nature*. Moreover, the update may include some meta-information to be used by the update mechanism to perform the update. They also defend the need to have some kind of *update manager* that controls the whole processs and is able to lead the system to a *quiescent state* before applying the update (see Section 5.1). This approach certainly reduces the transparency of the dynamic update from the point of view of both the programmer and the final user. However, on the other hand it simplifies the design of updateable software and allows to get better results. See Section 5.4 for references of other types of *intrusion* between the update mechanism and the managed application.

4 Additional Results

There are other papers that analyze some complementary issues to the dynamic software update problem but are not centered on updating mechanisms. They analyze how to parameterize the resulting consistency among different software pieces, the requirements set by different standards, etc. They are briefly discussed in the sequel.

4.1 *CONIC: an Integrated Approach to Distributed Computer Control Systems*, Kramer *et al.*, 1983

Kramer *et al.* introduce CONIC [47, 48], a configuration language used to declarative describe both the configuration of a software system and the changes to such configuration that may be dynamically applied.

CONIC uses plain-text configuration files to express the structure of the modules of a distributed system. Each module has some *input ports* which are *abstractions* that represent the input of data into a module and provides some *output ports* which are the corresponding abstractions to represent the output of data out of a module. CONIC provides some syntactic constructions to *connect* the input port of a module and the output port of another module. It can even allow one-to-n connections, under some constraints.

CONIC also provides a way to express *changes* to such a structural definition, like adding or removing modules or modifying the connection among ports. It also allows to *group* modules and assign modules or group of modules to physical nodes.

Once a CONIC specification is defined, it is *compiled* into some binary files. Change specifications are also compiled and the result is a set of operating system-dependent commands whose execution causes the desired change.

This kind of declarative configuration of the structure of an application reminds the idea behind some *Dependency Injection* and *Inversion of Control* concepts and technologies like Spring's Inversion of Control (IoC) [8].

4.2 *Towards Flexible and Safe Technology for Runtime Evolution of Java Language Applications*, Dmitriev, 2001

Dmitriev [29] studies the mechanisms to dynamically update Java programs included in the standard Java Virtual Machine.

For instance, the HotSpot Java Virtual Machine [27] has a mechanism to replace the Java bytecode in runtime, although it was devised to be executed in debugging time (by means of the Java Debugger Wire Protocol), as an aid to programmers, to speed their development and test processes. That mechanism only allows to replace the implementation of the methods. It does not allow other changes, like changes to the interfaces of the classes. Moreover, it requires some sort of *quiescent* state in which the methods to update are not currently being executed (see Section 5.1 for other definitions of quiescence).

Future versions of the mechanism will include new features like the possibility to add new fields, methods or constructors to existing classes or interfaces and deleting them as well as altering the type hierarchies (for instance, adding a class or interface to an existing type hierarchy or removing an existing one).

In Section 5.8 we list other references that study or use the facilities directly offered by other *underlying* technologies, like programming languages, their standard libraries or any other *base* level.

4.3 *Dynamic Service Update Based on OSGi*, Chen and Huang, 2009

Chen and Huang [26] study the problem of updating a service bundle, in the context of OSGi applications, that can be summarized as follows. In OSGi, services are usually defined by interfaces and it is possible to have several implementations of a given interface. Moreover, it is possible to dynamically update a service which is bound to a given name, by publishing a new version under the same name. From that moment, if a client *resolves* the name it gets a *reference* to the new version. The main problem to solve is that when updating a service, the state kept by the older version should be *transferred* to the new version (in Section 5.5 we review other issues related to the need to transfer existing states). There is a secondary problem regarding to the *previous* clients, that still have references to the older version of the service. To consider the update complete, the references should be somehow updated to *point* to the new version of the service.

They propose a first solution to solve both problems. First, the programmer must statically specify a *safe update point* in the source code of the bundles. This point must be chosen so when the control flow of the program reaches that point, it is guaranteed that the state of the bundle to replace is not being modified. The choice of such points can be done manually or by means of some kind of source code analyzer. When an update to a bundle is to be applied, the bundle is forced to transit to the *safe update point*, which actually can be seen as a *quiescent* state (see Section 5.1).

Once reached that state, a *state transfer* may be performed, in order to update the state of the new version of the bundle. Although it is not explicitly said, some *transformation functions* may be needed to adapt the data to the possibly different format used by the new version of the bundle. Other references related to state transfer and transformation functions are provided in Sections 5.5 and 5.6, respectively.

Finally, the old clients are forced to restart, so they can get references to the new versions of the updated bundles. Although this last step is a practical one, it nevertheless reduces the *transparency* of the solution.

The authors suggest a second solution that may offer a better transparency level, at least from the point of view of the application user. The solution consists in adding an indirection level, by means of *dynamic proxies* that would act as intermediaries among the clients and the service bundles. These proxies may *wrap* the current real references to the OSGi service bundles. If no updates have to be performed, the client invocations to the service methods are simply forwarded to the service bundles. When a dynamic update has to be applied, these proxies are notified and then, they *refresh* the references by disposing the old ones and getting new updated ones, thus hiding the update from the point of view of client code. In Section 5.3 we review other references that use different types of indirection.

See Section 5.9 for additional references about OSGi.

4.4 *Automated Versioning in OSGi: a Mechanism for Component Software Consistency Guarantee*, Bauml and Brada, 2009

In [19] the authors propose a procedure to automatically *deduce* the *version string* (e.g. something like "1.5.7") of an update. The version strings considered are strings of the form `a.b.c`, where `a`, `b` and `c` are the *major*, *minor* and *micro* version numbers, respectively.

As a starting point, a programmer has a given version of an application, whose version string is already known and the following version of the application, whose version string has not been chosen yet. The goal is to choose a proper string version for the new version of the application.

This has traditionally been done manually, according to some *semantic* criteria, choosen by the programmer. Instead, [19] shows a proposal to automate the election of the new string version. The idea is to sintactically analyze the source code of both a given version of the application and the following version. The procedure includes a number of *rules* to decide when to choose to alter the major, minor and micro version numbers.

This procedure is originally proposed in the context of the dynamic update of OSGi [7] applications (bundles), but it may be applied to any other application written in a regular current object-oriented programming language.

The authors admit that the proposed procedure fails to recognize *semantic changes* between a version of the application and the following one, so they cannot be considered to choose the new version string. They reject the use of *semantic analyzers* due to their excessive computational cost.

We may suggest the use of some *meta-data*, added by the programmer. The simplest way is to include some comments in the source code, with some specific syntax. If needed, a more elaborated mechanism could be used. For instance, in Java (since version 5.0), programs can include Java Annotations and they can be configured to be included in the bytecode generated by the compiler and retained by the Java Virtual Machine, so they can even be read and used in runtime. By means of any of these mechanisms, programmers may *suggest* changes to the major, minor or micro version numbers, according to semantic criteria (or any other criteria they may consider).

4.5 *An Application-Based Adaptive Replica Consistency for Cloud Storage*, Wang *et al.*, 2010

Wang *et al.* [75] propose an adaptive mechanism to 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 runtime, during the regular execution, according to the observed rates of read and write operations. Thus, the system may be in one of four modes, that range from a *relaxed* consistency mode to a *strong* consistency mode.

According to their proposal a replicated system may follow this organization. First, there is a *central node* called *master node*. Then, there are a number of *first-level* replicas called *deputy nodes* (they use 3 nodes). Finally, there are a number of *second-level* replicas called *child nodes*.

The idea is to ensure that the master node and all the deputy nodes are up to date at any moment. Depending on the current read and write rates, the transfer of the updates to the child nodes may be done immediatelly or deferred.

The authors identify four operation modes, depending on the read and write rates. For each mode, a read and write propagation scheme is proposed:

1. Case C=1: high read rate and low write rate. This is a *strong consistency mode*. A write operation sent to any replica is redirected to the master, which redirects it to all the replicas, thus achieving full consistency among all the replicas. When a replica receives a read, it can attend it directly.

2. Case C=2: high read rate and high write rate. This is a *trade-off mode*. A write operation sent to any replica is redirected to the master. Then, the master compares the timestamp of the write operation with the timestamp of the last write applied. If the difference is higher than a given threshold (this is, if the last write was applied *too long ago*), then it is forwarded to all the replicas (because they are considered to be *outdatet*). Otherwise, it is only resent to the deputy nodes. When a replica receives a read operation, it is attended immediately (so an *old* value may be read).

3. Case C=3: low read rate and high write rate. This is another *trade-off mode*, which tries to save bandwidth. A write operation sent to a replica is resent to the master, who resends it to all the deputy nodes. When the master or any of the deputy nodes receives a read request, it sends a response immediatelly. When a child node receives a read request, it resends it to the *closest* deputy.

4. Case C=4: low read rate and low write rate. This mode is similar to the previous one. The only different case is when a child node receives a read request. The child node compares the timestamp of the read request against the timestamp of the last write operation applied locally. If the difference is higher than a given threshold (this is, if the last write was applied *too long ago*), then the child node retrieves the requested value from the closest deputy node and returns it to the client.

In initialization time, the system is set in a strong consistency mode. Then a master node and the deputy nodes are chosen, according to their *geographic position*. Once the system is started, the read and write operations issued by the clients are counted and the mode is switched periodically, according to the rules shown above.

The mechanism considers the failure of the master node or a deputy node. If the master node fails, a deputy node is promoted to master. If a deputy node fails, a child node is promoted to deputy.

In Section 5.11 we provide some other references that cover some topics related to the dynamic update of replicated systems.

4.6 *Dynamic Software Update for Component-based Distributed Systems*, Panzica, 2011

In [52], Panzica proposes a classification of dynamic updates by means of an interface automata notation [28]. His classification includes the following classes in which a component C is replaced by a component C':

- Class 1: "Perfective Update". C' is a specialization or *subtype* of C.
- Class 2: "Corrective Update". C' is at the same time a) a specialization of C and b) an improvement of C.
- Class 3: "Partial Compatibility". C' offers only a part of the interface offered by C and in addition, it offers a new part.
- Class 4: "Incompatibility". C' es completely different to C.

The author uses a simulation framework to perform a process of validation to assess the *timeliness* and *disruption* produced by applying changes of the class 4 to an example application. The results are compared against the results got when applying the same changes to a system in which the components to be replaced must be quiescent [49] before being updated.

This classification and the subsequent analysis provided in [52] only considers the *sintactical view* of the application. No semantic concerns are taken into account.

5 Important 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.

5.1 Quiescence

A number of papers use some form of *quiescence*. The basic idea is that before updating some component, from a given version to the next one, 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.

One of the first works that shows an idea of *quiescence* is [49] by Kramer *et al.* (see Section 3.4), based on a previous idea from [46]. Informally, a node is *quiescent* if it is not going to start a data exchange or attending any data exchange with any other node. In [49], the authors argue that to apply an update that affects some nodes, they must be in a quiescent state.

Segal and Frieder [66, 33] use the concepts of *active process* and *inactive process* (see Section 3.3). They propose a criterion to decide when a process can be considered *inactive* and show that to apply a dynamic update to a process in a *safe* manner, it must be *inactive*.

Gupta *et al.* [40], propose a dynamic update mechanism that uses a similar idea. Before updating a function of a process, the execution stack is inspected to see if the function is present (*i.e.* if the funcion is *being executed*). The update can only be applied if it is not present (*i.e.* if the function is not being executed).

Chen and Huang [26], propose a mechanism to update bundles in an OSGi application that forces the updateable bundles to reach some *safe state*, in which none of the to-be-updated bundles is currently being executed.

Bannò *et al.* [17] also point out the need to lead the component to replace to a *quiescent* state, like in [49], to fulfill a requirement known as *consistency of control flow*. They also state that, in general, it will be necessary to guarantee some other properties, like some certain *semantic consistency* property. A way to reach such a quiescent state is to wait until the current requests and invocations end up running, *pause* the handling of incoming requests, apply the update and finally resume the handling of new requests.

Giuffrida and Tanenbaum [34] propose another update mechanism that uses an *update manager* component. When a dynamic update has to be performed, this update manager notifies the components to update. These transit, as soon as possible, to a *controlled* state (which is actually some form of *quiescence*, like in [49]), save their state in a persistence place and send back an answer. When the *update manager* receives all the answers, the update can be applied.

This idea of *stable status* or *quiescence* appears in many other references: [21, 18, 45, 20, 63, 15, 73, 43]. It can also be applied in other settings more or less related to dynamic software update but somehow different from the work referenced above. For instance, Dmitriev [29] talks about the dynamic update of methods of Java classes and the support offered by the HotSpot Java Virtual Machine (see Section 4.2). The mechanism is still under development, but it already offers some limited dynamic update mechanism, to ease the develoment 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*.

On the other hand, the *quiescence* concept and especially its *blocking requirements* have been criticized by some authors. For instance, Vandewoude *et al.* [74] argue that the *quiescence* concept in [49] 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.

5.2 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.

One of the first authors to propose the use of *binary rewriting* was Fabry [30]. He proposes the addition of a level of indirection (see Section 5.3) and the rewriting of low level binary instructions to update such indirection level.

Milazzo *et al.* [53] use some bytecode rewriting of Java classes to build an intermediary level that enables a regular Java application to be updated in runtime. Bytecode rewriting is also used to update the clients accordingly (see Section 3.13).

Hicks and Nettles [44] use some binary rewriting techniques to modify the service implementation, data types and the client code that accesses to the patched code (see Section 3.12).

Gregersen and Jørgensen [36] use the standard instrumentation facilities offered by the standard Java Virtual Machine as part of their mechanism of dynamic update of Java programs (see Section 3.20).

Bannò *et al.* [17] 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 [17] and Section 3.22).

Chen *et al.* [24, 25] propose their POLUS framework which is based on *dynamic patches* which are applied dynamically by editing the binary code (see Section 2.2 of [24] and Section 3.15).

On the other hand, there are currently available a number of tools and libraries that offer services related to bytecode manipulation (including runtime manipulations). For the Java programming language, there are many alternatives like ObjectWeb ASM [56, 22, 50], CGLIB [9], Javassist [11, 71], Apache Commons BCEL [32], Javeleon [10, 37], JRebel [77] and some others listed in [5].

5.3 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.

Fabry [30] is one of the first authors that propose the use of an indirection level, to be used in conjunction with some binary-level overwriting (see Section 5.2), that basically consists of jump instructions that perform the proper *redirection* of a call made to a service function from some client code.

Bloom's Ph. D. thesis [21], reuses the idea of redirecting the calls to the updateable code by *remapping* some handlers, in the context of Argus programs (see Section 3.2).

Segal and Frieder [66, 33] use *interprocedures*, which are some intermediary procedures used to *redirect* the client invocations to *old version* procedures to their *new version* counterparts (see Section 3.3).

Purtilo *et al.* [59, 58] 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 (see Section 3.5).

The proposal in [69] includes de use of some intermediary objects called *Service Facilities* (see Section 3.9). These intermediaries encapsulate the objects that offer the real service and typically get replaced whenever a dynamic update is performed. The indirection level offered by the intermediaries make the update transparent to the client code.

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. See Section 3.13 for additional details.

Ajmani *et al.* [13] 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.

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 (see Section 3.15).

Gregersen and Jørgensen [36] 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.* [17] also use some specific Java class loaders and some intermediary objects to control the execution of updateable components (see Section 3.22).

5.4 Intrusion and Cooperation

The dynamic update mechanisms proposed by some authors identify the need of or depend on some level of *intrusion*, thus making the managed programs and applications aware of the update mechanism. This *intrusion* can take a number of different forms.

For instance, Kramer and Magee [49, Sections III and IV.B] identify a double relationship between an update mechanism and the application to update: the former may need to invoke functions offered by the later and this may notify the former, for instance, about state changes (see Section 3.4).

Hicks and Nettles [44] propose a mechanism that allows the programmer to *mark* places in the code that should not be interrupted by a dynamic update (see Section 3.12).

On the contrary, in Ginseng, by Neamtiu *et al.* [55], the programmer can identify *safe update points* in the source code, in which an update may be safely performed.

Bannò *et al.* [17, Sections II.B and III.A] 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 (see Section 3.22).

Giuffrida and Tanenbaum [34] argue that the best approach to build dynamically updateable systems consists in making them aware of the dynamic update process and even ask the programmer to provide some meta-information to *help* the update mechanism (see Section 3.23).

Finally, the use of *state transfer functions* (see Section 5.5) and *transform functions* (see Section 5.6) can also be considered a type of *intrusion*.

5.5 State Transfer

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. Some of them use a variation of the idea proposed by Liskov and Herlihy [42]. 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 Section 5.6) and then transferred to the new version of the updateable item, by means of its `setState`-like function.

In his Ph.D. thesis, Bloom [21] identifies the need of transferring the *volatile* state managed by the part of the program to be replaced, to the new implementation (see Section 3.2).

Purtilo *et al.* [59, 58] 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 (see Section 3.5).

Gupta *et al.* [24] propose a dynamic update procedure that uses a state transfer step to transfer the state of the original process to its updated version (see Section 3.6). The procedures proposed by Solarski *et al.* in [68] and later in [67] for dynamically updating the replicas of a replicated system include a *state transfer* step during the update of the replicas (see Section 3.10). The proposal by Sridhar *et al.* in [69] (see Section 4.1) includes a *State Migration* mechanism to transfer the state from an old *proxy* object to a new one.

Gregersen and Jørgensen [36] also use a state transfer step, which has the special feature of being applied lazily, on demand (see Section 3.20).

Chen and Huang [26] include a step to transfer the state of an old version of an OSGi bundle to the new version in the procedure they propose to dynamically update OSGi applications.

Bannò *et al.* [17] 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.

5.6 Transformation Functions

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. This functions are typically provided by the programmer, like in [21, 33, 59, 44, 13, 70, 54, 26].

This topic is closely related to the use of *state transfer* functions (see Section 5.5).

5.7 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.

For instance, Stoyle *et al.* [70] and Murarka *et al.* [54] propose the static analysis of the source code to identify points in which it is possible to dynamically apply updates to the classes while ensuring some *correctness* property. For additional details, see Sections 3.16 and 3.19, respectively.

Neamtiu *et al.* propose Ginseng [55], a dynamic update solution for programs written in C. In their solution, they depend on some static analysis of the source code to ensure that the updates are *type-safe*. Moreover, they also use annotated source code to identify *safe* points in which the update can be applied, as in the previous references, although in this case, these points must be explicitly marked by the programmer. Altekar *et al.* propose OPUS [16] also depend on a similar analysis to detect *unsafe dynamic updates*.

Other authors, like Hicks and Nettles [44] 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 [19] 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 (see Section 4.4).

5.8 Using Underlying Facilities

A number of authors base their proposal on features of a given programming language or infrastructure.

For instance, in his Ph. D. thesis, Bloom [21] 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 [39], Ginseng by Neamtiu *et al.* [55] and POLUS by Chen *et al.* [24, 25].

Regarding the Java programming language and Virtual Machine, there are a large number of references. First, some authors propose DSU solutions for Java programs (for instance, [63], [53], [36], and [17]). Dmitriev [29] studies an existing mechanism available in the HotSpot Java Virtual Machine to allow the dynamic update of Java code in debugging time (see Section 4.2). Some other authors like Gregersen and Jørgensen [38] propose DSU mechanisms based on modifying the standard Java Virtual Machine (which presents a number of drawbacks, as identified by Bannò *et al.* [17]). 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 5.9 for additional references about OSGi).

Other authors propose solutions that are a bit more general and can be used with programs written in *imperative languages*, like Hicks and Nettles [44].

There are also some authors who develop their proposal based on their own infrastructure. For instance, Kramer and Magee base their proposal [48, 49] on their CONIC configuration language and infrastructure [47]. In Proteus, Stoyle *et al.* [70] describe a DSU solution based on its own programming language and compiler and runtime, among other tools and resources.

At a lower abstraction level, the solution proposed by Frieder and Segal [66, 33] needs that the hardware architecture of the undelying machine offers an *indirect addressing mode* (available in the Motorola MC68020 processor and the Intel's 386 architecture). Gupta and Jalote's proposal [40] 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 available in Motorola's 68020 microprocessor, but actually available in other microprocessors).

5.9 OSGi

OSGi [7, 14] is a platform to build Java applications from a number of modular, reusable and collaborative components (called *bundles*), that can be dynamically reloaded. A short introduction to OSGi can be found in [72].

There are a number of implementations of OSGi:

- Apache Felix [31], open source, by the Apache Software Foundation.
- Concierge [1], open source, especially designed for resource-constrained devices (see also [61]).

- Equinox [3]
- KnopflerFish [4]
- Oscar [6]

Moreover, there are some other proposal that extend OSGi or are related to OSGi in some way. For instance, Rellermeyer *et al.* propose R-OSGi [60], an extension of the standard OSGi specification to be used to build distributed systems (see Section 3.17). Another alternative also focused on distributed and cloud systems is OSGi*4C* [65] (see Section 3.21).

In [26] a proposal of a mechanism to dynamically update the bundles of an OSGi application is provided (see Section 4.3).

5.10 Version coexistence

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 5.1), performing the update and *uninstalling* or otherwise preventing both versions to run at the same time.

On the other hand, there are systems that allow such multiple version coexistence. Thus, clients of such multiple versions may execute concurrently, at least for a transitional interval that will end when all clients are also updated, using then the interfaces of the latest version.

For instance, in the context of *dynamic updating* of functions and procedures, Segal and Frieder [66, 33], 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 *ilusion* that different versions of the same procedure coexist. See Section 3.3 for additional details.

Ajmani *et al.* [13, 12] 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. See Section 3.14 for additional details.

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 [29] elaborates on different policies that may be implemented in the context of the dynamic update mechanism included in the Java Virtual Machine (see Section 4.2).

5.11 Replication

Few papers have tackled the topic of applying dynamic updates to replicated systems. In this section, some of them are surveyed.

For instance, Solarski and Meling [68] propose a procedure to dynamically update a distributed system that uses *active replication* (see Section 3.8). This work is later extended by Solarski [67] in his Ph.D. Thesis, by adding a procedure applicable to systems that use *passive replication* (see Section 3.10).

Wang *et al.* [75] propose a mechanism to dynamically change the consistency mode used by the replicas of a replicated system, depending on the observed rates of read and write operations issued by the clients (see Section 4.5).

5.12 Scheduling

In some of the dynamic update mechanisms surveyed, the programmer is allowed to decide about the *scheduling* of the updates.

For instance, in Hicks and Nettles [44], the programmer can decide when an update has to take place (see Section 3.12).

Another different example is [13], in which Ajmani *et al.* propose a number of *scheduling functions*, the programmer can choose one from, to update the nodes of a distributed system.

5.13 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* the update, just applying a patch to *update* the program from version $v+1$ to v (see Section 3.15).

Gregersen and Jørgensen [36] also allow the rollback of updates in their mechanism of dynamic update of Java programs (see Section 3.20).

6 Conclusion

This paper reviews a set of references related to the dynamic software upgrading problem. The requirements being considered in these systems are identified and briefly discussed. A chronological review of multiple solution proposals is given in Section 3, identifying different mechanisms that have been combined in some of the reviewed papers. Such mechanisms are described in more detail in Section 5.

As a result, this work provides a basis to start the analysis of these mechanisms in order to design and develop other frameworks for dynamic software upgrading. This field will require more efficient solutions for the elastic applications to be developed onto IaaS and PaaS cloud systems. To this end, the current paper provides a convenient set of references that explain those mechanisms.

References

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

[2] CVS (Concurrent Version System). http://www.cvshome.org/.

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

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

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

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

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

[8] Spring Framework. http://www.springsource.org/.

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

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

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

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

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

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

[15] 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.

[16] 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.

[17] 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.

[18] 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.

[19] 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.

[20] 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.

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

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

[23] Jim Buckley, Tom Mens, Matthias Zenger, Awais Rashid, and Günter Kniesel. Towards a Taxonomy of Software Change. *Journal of Software Maintenance and Evolution: Research and Practice*, 17(5), September 2005.

[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] Oracle Corporation. The HotSpot Java Virtual Machine, 2012. http://openjdk.java.net/groups/hotspot/.

[28] Luca de Alfaro and Thomas A Henzinger. Interface Automata. In *8th European Software Engineering Conference*, ESEC/FSE-9, pages 109–120, Vienna, Austria, September 2001. ACM.

[29] 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.

[30] 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.

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

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

[33] 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.

[34] 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.

[35] Li Gong. JXTA: a Network Programming Environment. *IEEE Internet Computing*, 5(3):88–95, May-June 2001.

[36] 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.

[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] 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.

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

[40] 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.

[41] 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.

[42] 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.

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

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

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

[46] J. Kramer and R. J. Cunningham. Towards a Notation for the Functional Design of Distributed Processing Systems. In *IEEE International Conference on Parallel Processing*, pages 69–76, August 1978.

[47] 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.

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

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

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

[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] L. Dailey Paulson. Computer System, Heal Thyself. *Computer*, 35(8):20–22, August 2002.

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

[59] 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.

[60] 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.

[61] 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.

[62] Jan S. Rellermeyer, Michael Duller, and Gustavo Alonso. Consistently Applying Updates to Compositions of Distributed OSGi Modules. In *1st International Workshop on Hot Topics in Software Upgrades*, number 9 in HotSWUp '08, Nashville, Tennessee, USA, 2008. ACM.

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

[64] Don Roberts, John Brant, and Ralph Johnson. A Refactoring Tool for Smalltalk. *Theory and Practice of Object Systems*, 3(4):253–263, October 1997.

[65] 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.

[66] 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.

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

[68] 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.

[69] 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.

[70] 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.

[71] 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.

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

[73] 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.

[74] 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.

[75] 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.

[76] Stefanos Zachariadis, Cecilia Mascolo, and Wolfgang Emmerich. The SATIN Component System-A Metamodel for Engineering Adaptable Mobile Systems. *IEEE Transactions on Software Engineering*, 32(11):910–927, November 2006.

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