TEM Journal. Volume 9, Issue 1, Pages 117-128, ISSN 2217-8309, DOI: 10.18421/TEM91-17, February 2020.

Challenges in Dynamic Software Updating

Danijel Mlinarić

Department of Applied Computing Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia

Abstract **– Changes in the domain or simple bug corrections lead to software updating. In general, software updating involves halting the running program, which is not desirable in the current applications, for reasons such as costs in business or undesired situations in mission-critical applications. To avoid program halting and to replace the current program version while running, many different Dynamic Software Updating (DSU) solutions have been developed. This paper presents a set of challenges in the existing research of dynamic software updating and the ideas for further developments.**

Keywords **– on-the-fly modification, hot-swapping, dynamic software change.**

1. Introduction

Software changes over time because of the changes or corrections in its functionalities. As a result of these changes, new software versions are produced. Replacing the current software version with the new one is referred to as software updating. Early distributed software systems, such as the airline reservation system, required both software availability and functionality changes [9].

DOI: 10.18421/TEM91-17
https://dx.doi.org/10.18421/TEM91-17

Corresponding author: Danijel Mlinarić,
Department of Applied Computing Faculty of Electrical Engineering and Computing, University of Zagreb, Croatia.
Email: danijel.mlinaric@fer.hr

Received: 06 January 2020.
Revised: 02 February 2020.
Accepted: 07 February 2020.
Published: 28 February 2020.

© 2020 Danijel Mlinarić; published by UIKTEN. This work is licensed under the Creative Commons Attribution-NonCommercial-NoDerivs 3.0 License.

The article is published with Open Access at www.temjournal.com

These two requirements are contrary to the classic updating process with the steps: halt, deployment of the new version and software restart. Therefore, it is difficult to balance high availability with frequent changes.

Dynamic software updating has been in the focus of many research studies. However, there is no agreed definition within the research community. Briefly, software environments that provide high availability with a support for changes at the same time are known as Dynamic Software Updating (DSU) [10], [15].

A wide range of applications and systems could benefit from the DSU. In *embedded systems*, e.g., for traffic lights, availability is highly needed, also as providing a possibility for changes that support new control strategies used by smart traffic control in cities. *Cloud systems* are required to provide high availability in PaaS (Platform as a Service) and Saas (Software as a Service) services due to end-user agreements, such as Service Level Agreements (SLA) [30]. In the case of the downtime, the service provider compensates users for availability [22]. In *operating systems* (OS), a restart of the system can negatively affect the working process of the end-user. Client applications connected to other entities such as web servers and remote databases, update due to changes in those entities, which can lead to lost data. In mobile applications, for example, an update of communication application disrupts end-users possibility to communicate to other users during the update. In business web applications, in which the application is a part of the business process, non-availability causes a stoppage of work and creates expenses for the end-users. In all these applications and systems, there is a struggle between availability and support for changes. To provide availability and correct behavior of software, DSU is required to overcome these problems.

This paper, based on the related literature, outlines the main requirements, characteristics, and implementation details to consider when using, improving or creating a DSU. Furthermore, an overview of possible progress is given based on the current DSU challenges. Section 2 of this paper

describes the main DSU requirements and their relationships. DSU requirements lead to specific updating problems described in section 3. Section 4 presents updating techniques and mechanisms to resolve DSU problems. Section 5 discusses DSU implementation regarding the software environment and DSU features. A set of challenges related to DSU is given in section 6. The last section is the conclusion.

2. Dynamic Software Updating Requirements

Up to date, many different techniques and approaches have been developed in accordance with different systems and applications. All DSU systems have requirements that are the same regardless of environment features and constraints. With the already mentioned availability and change support, they are:

- *Availability* – performing dynamic software updating does not affect the availability of the software, or it is not noticeable by the end-users [11].
- *Correctness* – updatable software should preserve the correct behavior during runtime [11]. Dynamic updateability does not affect the correct execution of the software.
- *Flexibility (Changeability)* – the set of possible changes between two versions of the software should be as large as possible [11]
- *Performance* – the increase in demand for system resources (processing power, memory, storage), in order to support DSU should be minimal [11], [29].
- *Simplicity (Usability)* – DSU usage should be simple and accessible [11], [29].

These goals are contrary to each other. It is challenging to provide a very large set of possible changes and simultaneously provide minimal usage effort for the developer while preserving the minimal performance cost. Almost every DSU approach balances between these requirements. On the other hand, availability and correctness should be viewed together. E.g. considering that the software is running without disruption, before, during and after the update but the program execution during or after the update is corrupted. This kind of software behavior is not desirable.

3. Related Problems

In order to achieve requirements stated in section 2, DSU faces several problems comparing to the classic software update scheme in a form halt, deploy, and restart. Updating an active program code at the unsafe moment can cause unpredicted software behavior. For example, presume that the dynamic updating is performed in the middle of the function after the variable declaration. A function in a newer version after the point of updating is executing computation instruction comprising the newly introduced variable that is undeclared in the previous version, which leads to the fault state. Another problem is to ensure smooth end-user experience during the update. DSU is required to provide transparency in a way that a component of the previous version can use a component in the new version. When DSU updates the entire program at once, this kind of problem is not an issue because all components are upgraded at once. On the other hand, the replacement of the whole program often leads to a long time of update, degrading availability requirement. Moreover, the whole program replacement is often followed with issues of correctness since the current state of the program can be lost. During the running application update, the DSU is required to preserve the current state, to support correctness and consequently to provide transparency to the end-users. To preserve the program state multiple program versions can exist simultaneously. If there are multiple versions of the same program object in the system, for example, a class instance in memory, it increases the demand for resources, which degrades system performance.

Many DSU approaches resolve these problems, which distinguish the DSU update process in comparison to the classic update process. Other possible problems are often arising from the DSU implementation related to the different software environments described in section 5.

4. Updating Techniques and Mechanisms

DSU related problems described in section 2 can be resolved with appropriate techniques divided into several categories shown in Table 1: *level of update, update of dependent components, time of update, state transfer*, and using *cleaning* and *rollback* mechanisms. *Level of update* determines which parts of the software need updating. For example, a single component or whole program can be updated. In dynamic updating, the dependency between components when components are of different versions is handled with an *update of dependent components. State transfer* handles the state conversion from the previous to the new version in terms of system correctness and end-user level transparency. How to proceed with updating the active program code is one of the questions

addressed by the *time of the update*. Timing techniques determine the time of update in different parts of the software system, such as in the distributed system. On the other hand, the system demands increase with dynamic updating, and the proper *cleaning* of the unused previous version memory fragments maintains the system availability and performance. During the process of the dynamic update, errors may occur, and there is a need for the possibility to *rollback* changes to the previous version in order to maintain system availability and correctness.

4.1. Level of Update

DSU can be designed to replace the whole program at once or to replace a single component, such as class [11], [37], class members [11], method [5], [19] or instruction [26]. To reduce the time of update, smaller components of the update are chosen, for example, instruction-level as presented in DynSec [26]. Such a size of the unit is appropriate for security patches. The change of a single instruction can fix the bug, e.g., buffer overflow. Components are the common level of update in various approaches to keep the update duration and performance overhead as minimal as possible compared to the whole program.

4.2. Update of Dependent Components

Fabry in [9] introduced the indirection level for dependent components to resolve problems when dependent components are in different program versions. Figure 1 shows the indirection level between the component versions [29].

The connection between components is duplex because components have both input and output. The indirection level has built-in logic for handling the connection between the previous and the new component version. The simple approach is to use a jump or jump like instruction as in [3], [10]. Frieder and Segal [10] use a special segment register containing the address of the procedures lookup table. However, such an approach depends on the CPU architecture. Other approaches of indirection level can be: *proxies* redirecting calls to new versions [11], *wrappers* - new version 'wraps' the previous [27], *pointers* - updating pointers from the previous to the new version objects [15], [37]. *Dynamic Proxification Framework* (DCF) [11] uses the *byte-code rewriting technique* to insert the code in the previous component version, which then calls the new version of the component.

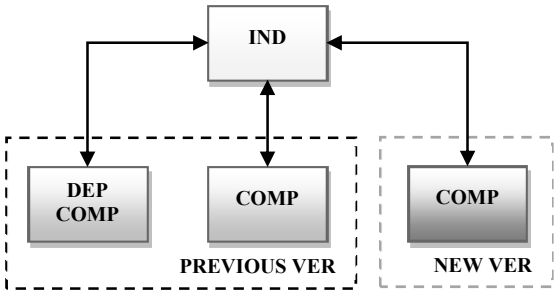

Figure 1. Indirection level between previous and new version of components (IND. – indirection logic, DEP COMP – dependent component, COMP – component, VER – version)

Dependent components can be dependent statically [11], [31], determined in static state, e.g., in software code or semantically, in runtime [10]. Static dependency can be determined automatically, whereas semantic dependency is defined by the programmer or detected in a higher-level analysis. Both types of dependency between components are required to provide the correctness of the DSU. The most common example of both dependency types is a software system wherein communication is performed. Communication program consists of two functions: for sending and receiving messages. Changing one function requires changing the other; otherwise, the function for sending the message sends the message in the previous version, but receives it in the new version. The receiving function cannot receive the message properly due to the change between the versions, which leads to incorrect behavior of the communication process. Such cases are solved with the use of programmers defined semantic dependencies list [10].

4.3. Time of Update – Safe Point of Update

The point in time when it is possible to make an update is important to preserve running software correctness. In [14], the authors proposed three categories of update timing control regarding the update of active functions: *activeness safety (AS), con-freeness safety (CFS),* and *manual identification*. *Activeness safety* prevents the update of active functions. *Con-freeness* [31] allows the update of active functions when it is type-safe. An update is type-safe when the block of code after the point of the update is not affected by the update. To ensure the correctness, both AS and CFS requires code modification, to a greater extent than the programmer's *manual identification* [14]. *Manual identification* relies on the programmer-defined safe points of update [3], [4], [14]. In single-threaded applications with one loop,

Table 1. DSU techniques by categories

level of update		dependent components		point of update		state transfer	
component	class [11], [37] method [5], [19] class members [11] instruction [26]	indirection	proxies [11] wrappers [27] pointers [15], [37] lookup tables [10]	safe points	AS [14] CFS [31] manual-identification [3], [4], [14] not handled [6], [11]	handling	automatic [3], [4], [11], [37] programmer defined [10], [22], [36]
		type	static [11], [31] dynamic/semantic [10]	delayed	quiescence [10], [17], [28] tranquility [8], [34] relaxed-synchronization [23] lazy [36]	type	lazy [3], [7], [9] first access (lazy) [11], [35] atomic [32], [37] bidirectional [3], [7], [35], [36]

the point of the update is suggested to be at the end of the loop, where there are no active transactions [29], [37]. On the other hand, defining the points of the update is complex in multithreaded programs. The programmers are required to follow programming patterns because of synchronization problems, such as a suggestion for the loops. In *First-Class Context* [36] approach, the time of update is not exactly determined, and it is called *incremental update*. In the previous version, active transactions and requests are running until the end of the transaction or request. After the update, every new transaction or request is running on the new version. It is a lazy update from the previous to the new version. Some approaches reject the update immediately if functions which are to be updated are active i.e. exist on the call stack or do not handle update timing at all [6], [11]. Others, like *quiescence* [17], *tranquility* [34], *relaxed synchronization* [23], handle updates with delay. A brief description of each of them is as follows. First, assume that in DSU, depending on the application and system type, we have entities of dynamic software updating environment affecting the correct running and the behavior of the system, e.g., in procedural programming languages these are functions, nodes in distributed systems, processes in operating systems, transactions in databases and communication systems.

1) *Quiescence* [17]. To perform the operation of the dynamic update of an updatable entity, the entity has to be inactive. Further on, all entities reachable by an updatable entity have to be inactive, and other entities that might reach the updatable entity will be inactive during the update. Such a concept has been used in [10], [17], [28].
2) *Tranquility (relaxed quiescence)* [8], [34]. It is similar to *quiescence*, but it does not require the entities that can reach updatable entity to be strictly inactive during the update. Instead, both the entities connected to the updateable entity and the updateable itself are inactive and will both remain inactive during the update process. The connected entities will not be active when the updateable entity is inactive.
3) *Relaxed synchronization* [23]. Assume that in the program code, there are points of update that are equivalent, which means it does not matter where in the program code, the update will occur. The equivalent update points create a block of code where the update does not affect any of the block statements. If the update request is received in the middle of the block, program (thread) execution can continue until the end of the block, and the update will then be executed.

4.4. State transfer

A simple prerequisite by some approaches is to assume that the new version can use the state of the previous version [6], which mostly induces reduced changeability. There are two basic categories: *automatic conversion* and *programmer-defined conversion*. *Automatic conversion* [11], [37] copy objects data from the previous to the new version. It includes, in some cases, *type conversion* in terms of structural changes, also as primitive types change with associated value conversion, e.g., a number to string conversion. Moreover, if some of the structure is changed, for example, in adding the field to the class, default programmer-defined values are used. Manual approach [10], [22], [36] includes programmer written transformation functions from the previous to the new state. This approach extends flexibility when a major change between versions occurs, but also, at the same time, it disrupts programming transparency. Writing a custom transformation code often requires that the programmer uses new language constructs and follow some conventions or design patterns.

Another aspect of transformation is the time when it occurs, immediately or after the incoming update

request. A popular technique for performing state transformation after receiving the update request is on-demand [3], [7], [9], also called a *lazy transformation*. The value of the field is transformed on the first access, as seen in [11], [35]. Lazy transformation is recommended because it has a good impact on the performance and duration of the update. On the other hand, transformation functions can be bidirectional [3], [7], [36] to maintain synchronization between the multiple existing versions. The change in one version affects the value of another.

4.5. Cleaning

In higher-level languages, DSU relies on the *garbage collector* mechanism. When old components are no longer used, garbage collector marks them for cleaning [32], [37]. In [36], when the old context is not in use, it is finalized. The finalization process migrates all old context objects to the new context and after that, the old context is garbage collected.

4.6. Rollback

In case of errors during the update, in order to support the availability and correctness of DSU, the system is required to provide a *rollback* feature [3], [11]. However, *rollback* cannot be done in environments without reversible actions. In general, *re-executing* and *rollback* are mechanisms by which the program execution is rolling back to the first point where the previous and the new software versions are equal [13]. POLUS [3] supports reversible patches that convert the currently running to the previous version. Rollback may be designed as a part of multiple versions existence during the update when there is a need for the rollback in case of exceptions.

5. Developing Dynamic Software Updating

Since DSU implementation largely depends on different environments and usage, there are different restrictions and features to consider. Several major are described and shown in Table 2: *programming language* defines implementation platform, *type of application* and *runtime environment* determine architecture of DSU, *type safety, defining set of changes, concurrency* and *coexisting of multiple versions*, which are implementation details.

5.1. Programming Language

Some DSU concepts define custom programming language to support dynamic upgrade capability, such as in [7], [13], [31]. This kind of DSU requires programmers to learn new programming languages. Furthermore, such DSU supports a large set of changes and long-term usage simplicity because the programming language is developed with dynamic updating features in mind. The other approach is to use a programming language popular among the developers, such as C, C++, Java [33], like in [3], [11], [37]. Using popular programming language without modifications can increase DSU implementation complexity due to a lack of dynamic updating features in the programing language. Using a less popular programming language as in [9], [36], is often the case when the environment, e.g., legacy or system constraints, do not allow any other solution.

5.2. Application Type

Because of the environment properties and the purpose of the system, implementation depends on the application type. DSU for an embedded environment [25] is more likely to be simpler to implement. Embedded software has a smaller memory footprint, meaning the translation of the state is simpler for handling. Object-oriented programming languages are not often used for embedded systems as opposed to business applications, which increases the demand for system resources. Furthermore, current business applications run on a web [3], [14], [31] or cloud systems [1], [22]. These systems can be distributed due to load balancing (techniques for uniformly balance load over each node) and serving a large set of users, meaning that DSU implementation performs synchronization between multiple nodes [1]. Standardized business applications consist of three layers: database, server, and client. Dynamic update of business application on a three-layer architecture involve a separate update of each layer, e.g., database [9], [18], with the synchronization mechanism between these layers [1]. In DSU handling operating systems, the time of the update can be simpler to detect since function in the process can be blocked instead of inactive [29]. The linux operating system provides 'hook' *(ptrace* function) for easy injection of code into the running application [3]. Built-in mechanisms as *ptrace* simplify the implementation of DSU supporting the dynamic update of OS modules. Native i.e. client applications regarding DSU [6], [11], [37] can perform updates during an inactive state similar to OS blocked processes. If applications run on the intermediate-language platforms, they can be dynamically updated using existing DSU techniques on platforms such as

Smalltalk, Java on VM, C# on .NET Common Language Runtime - CLR.

5.3. Runtime Environment

There is a difference between implementing long-term and short-term DSU. Current short-term DSU approaches are used for development environments such as IDE (*Integrated Development Environment*) as in [11], [37]. On the other hand, long-term DSU would be used in the production environment. Although various approaches in the existing research solve many DSU runtime challenges, there is no research on the long-term running DSU for the production environment.

In the development environment, DSU is a tool to aid software debugging activities such as refactoring or bug correction. Current commercial level available solutions are *HotSwap* [5], [6] in Eclipse IDE, and *Edit* and *Continue* [19] in the Microsoft Visual Studio IDE. Such IDEs support method body changes without changing the method signatures or complex types, e.g., in object-oriented languages, adding or removing the class members. The programmer can change or add active method instruction while debugging the same method, mainly to correct bugs. The change immediately affects the functionality of the program and further execution.

In [11], [37] authors extended the set of changes, when comparing to available IDE mechanisms, enabling the change of class hierarchy and adding or deleting members of classes. In such cases, binary compatibility has to be ensured, which is discussed in subsection 5.4. Otherwise, the proper program execution is corrupted.

The techniques and methods developed for the debugging environment could be applied in a production environment, although with modifications due to differences in these environments. The debugging environment provides simple detection of the point of update and simplifies the state transfer process. Breakpoints in the debug mode are the points of the update, which are points in a program that can suspend execution. On the other hand, production environments increase the complexity of the implementation due to continuous execution.

5.4. Type Safety – Binary Compatibility

If the DSU level of the update is smaller than the replacement of the whole program, the update of the dependent component handles type safety. It is particularly important in approaches relying on static programming languages because of the static type checking. Changing the component interface is a modification of the component interface signature, i.e., type change. The type between dependent components is required to be equal before and after the update. One of the first approaches using a static type checking was DYMOS [4] using StarMod distributed version of the Modula programming language. Approaches that do not support signature change do not have a requirement to handle type safety [6], [16]. Other approaches, with the support for interface change, perform static check [4], [11], [37] or use functions to provide type safety [3], [7], [15]. In DSU with a static check, the updates that fail type safety checks are rejected to maintain the correctness. Conversion functions convert calls between previous versions of the dependent component to the component in the new version. Such functions are provided as a part of the update defined by the programmers.

DSUs with a convention that updates are required to be type-safe as [11] rely on the fact that in the classic update scheme, the programmers compile changes before the halt, redeploy and restart of the program. Compiling ensures the type safety, i.e., binary compatibility of the new version. In general, adding new objects such as classes, methods, fields, and variables into the code is binary compatible operation. On the other hand, removing the objects is considered an incompatible binary operation i.e. type unsafe. Changing the body of functions or methods is also binary compatible change, and widely supported in debugging environments described in subsection 5.3. Binary incompatible changes can be handled with coexisting of multiple versions and with the use of predefined DSU configuration as in [37]. Introduced binary incompatible configurations are: *static check* - if deleted object is accessed in further program execution, update is rejected, *dynamic check* - search for reference during runtime, returns an exception on fail, *access deleted member* - supports accessing an old, deleted method or static field, whereas for other deleted objects returns exception, *access old members* - method in previous version access the old version of the method instead of the new version i.e. version consistency [37].

Furthermore, in databases, type safety consists of data scheme comparing between the previous and the new version. In [22] scheme comparing is called a *safety check,* which ensures that accessing the modified object, in this case, the database table, is always performed on the new scheme version. There are two cases, presuming that Δ is a set of all differences in the scheme between versions. Differences are: added, deleted, or modified table attributes. Safety check before accessing the table *T* performs the intersection

between Δ and the scheme of table S_T. In the case of the empty intersection $\Delta \cap S_T = \emptyset$, the table is accessed as there are no changes. Otherwise, $\Delta \cap S_T \neq \emptyset$ and access is delayed as long as DSU is performing the migration of the scheme and existing data from the previous to the new version, corresponding to *lazy transformation*.

Dynamic languages such as PHP, JavaScript, and Ruby do not have explicit type declaration, as types are handled dynamically. This feature can simplify the implementation of DSU because it does not have to handle type safety as in static programming languages.

5.5. Concurrency

Multithreading or concurrency support in DSU is closely connected with the time of update described in subsection 4.3. Some approaches [6], [7] do not consider multithreading, even if their handling of update timing can provide multithreading support [6], [7]. Other approaches that support multithreading like [11] advise programmers to use programming patterns and practices in multithreaded programs. These are common when developing multithreaded programs because developing multithreaded applications is often complex due to synchronization between threads. DSU that includes multithreading support should avoid the occurrence of deadlocks. DSU can be implemented by using proper technique for the time of update and modifying system calls, e.g., with *indirection*, used for synchronizing threads. In [11], the Java environment provides *wait* and *notify* synchronization functions. The former is used for blocking the execution of the thread and the later to release the waiting thread. The problem arises when in the previous version of the object, function *wait* is called, causing thread waiting, and after the update calling of the *notify* function is only performed on the new version of the object. The thread in the previous version will never end because it will never receive unblock instruction. The changes between versions in program code handling synchronization like unblocking the thread should be done with caution. Changed functionality in the newer version of such objects can leave threads waiting, causing deadlocks [11]. These changes are not compatible with dynamic updating.

5.6. Defining the Set of Changes – the set of differences

The difference between versions can be: determined automatically by static analysis tools on a code level, defined by the programmer or determined at the higher level of abstraction or design level [3], [13], [25]. Code level can be object code e.g. byte-code level [16] or source code level [15], [32]. However, the source code of previous versions is not always available. In such cases, object code has an advantage over the source code. Furthermore, comparing on the lower level may introduce unnecessary updates of negligible changes. On the other hand, comparing small changes in the case of optimization or bug correction, on a higher level can cause update rejection because no changes are detected [29]. E.g., single changed instruction where the size of an array is defined can be neglected when comparing program versions on the higher level using CFGs (*Control Flow Graphs*) [25]. Lower level automatic comparing is simpler to implement than the automatic comparing on a higher level, such as class hierarchy. Automatic comparing relies on static analysis and often is implemented as a separated tool that produces differences in a format known in advance, custom [32] or particular language such as XML (*eXtensible Markup Language*). In [26], the difference is produced by a special tool comparing the differences of the functions binary code. Furthermore, software versions can be stored in the repository such as a database, in which an update to a specific version is determined by repository analysis [2]. From the point of implementation, differences defined by the programmer as part of an update are the simplest case, but it degrades the simplicity goal.

5.7. Coexisting of Multiple Versions

It is desirable to have multiple versions at the same time, e.g., in cases of cyclic dependencies, when in one version class depends on another, but in the next version, the second class depends on the first [37]. Multiple version of the same object in memory increases memory demands and affects the correctness. The correctness may be degraded because DSU provides *version consistency* [24], meaning that dependent objects are required to be compatible. If multiple versions of the same object exist, it is challenging to ensure state synchronization between them. In distributed systems, there can be more than two versions, which increase the complexity of synchronization between the system entities.

Table 2. DSU implementation details

<table>
<tr><th>programming language</th><th>application type</th><th>runtime environment</th><th colspan="2">type safety</th><th colspan="2">set of changes</th></tr>
<tr><td rowspan="2">custom
[7], [13], [31]
popular
[3], [11], [16], [37]
rare
[9], [36]</td><td rowspan="2">embedded [25]
web/cloud/server
[1], [3], [14], [31], [32]
OS [3]
native/client
[6], [11], [37]
database
[9], [18]</td><td rowspan="2">short-term (dev.)
[5], [6], [11], [19], [37]
long-term (prod.)</td><td>interface change</td><td>static-check
[4], [11], [31], [37]
conversion-functions
[3], [7], [15]
not supported/handled
[6], [16]</td><td>detection level</td><td>object code [16]
source code
[15], [32]
higher level
[3], [13], [25]</td></tr>
<tr><td>binary comp.</td><td>compatible [11]
incompatible [37]</td><td>format</td><td>custom [32]
repository [2]</td></tr>
</table>

6. DSU Related Challenges

Many software applications in a distributed environment consist of multiple layers and nodes.

Figure *2* shows a web system based on three layers: database, server and client, with multiple nodes.

Although layers are presented as separate entities, they can be on one physical place as a single node. In real-world situations, current business applications have physically separated layers. The database layer can be distributed on multiple servers, or consisted of a main database server with the secondary failover server. The main application logic layer often consists of multiple nodes, i.e., servers with load balancing. Clients are various devices, such as desktop computers, mobile or IoT devices. The client-side of the application usually resides on the internet browser but can be the native application. DSU logic can be placed depending on the implementation: inside the IDE, distributed in each node, or on a single physically separated location.

Having an entire system of current web applications in mind, we can consider several dynamic software updating challenges, from IDE enhancements, mobile devices, cloud, and distributed systems to decision making analysis for improving DSU correctness and availability. At last various benchmarks are required for DSU evaluation.

Considering the requirements of the real production systems and related literature discussions, in following subsections, challenges, and possible future work related to dynamic software updating is discussed.

6.1. Enhancements in the development environment

DSU concepts applied in the debugging environment could also be applied in the production environment.

However, there is a lack of research on deploying dynamic updates in the production environment. Existing DSU with further research and new DSU approaches can provide developers in the IDE environment the ability to change code on the fly when debugging and deploying new version with dynamic updating to the web, distributed or embedded distributed system shown in

Figure *2*. More research is needed to develop further DSU enhancements related to connecting IDE environments to the production environments.

6.2. DSU Methods and Techniques for Mobile Devices

Considerable research has been done in Java that opens the possibility for research in dynamic updates of Android Java-based applications. Such implementation needs to count on using the Android VM for Java, which is different from the standard Java VM. Another example is iOS applications that are commonly developed in Objective C/C++ programming language similar to standard C and C++. There are many DSU approaches based on C, C++, or C-like [3], [31] programming languages extending the possibility of applying them on mobile and IoT environments with limited resources and power consumption restrictions [29].

6.3. Applying DSU to Cloud and Web Applications

In the research literature, in terms of dynamic software updating, cloud and web applications are not commonly found. Bhattacharya and Neamtiu in [1] considered the problem of each layer separate dynamic update validation and synchronization between

multiple nodes. There is a lack of research beyond problem discussion related to dynamic updating managing and synchronization between multiple layers. DSU is required to consider consistency between nodes and layers, including possibly different versions in nodes. A situation where the system contains different versions across nodes can appear temporarily during an update, or intentionally when different groups of nodes are serving different clients. For example, the development system can contain a test instance. Therefore, before deploying to the production environment, a new version is deployed to the test instance. Distributed and multilayered systems require an appropriate centralized manager to analyze the current system state and to perform dynamic updating across layers and nodes. It is challenging to determine and design parts of such a centralized updating manager handling different versions in nodes and layers.

On the other hand, there is a lack of research on dynamic updating on the web client-side. Client-side consists of internet browser script (JavaScript), page structure (HTML), presentation description (CSS – *Cascading Style Sheet*) and internet browser local storage. Scripts and local storage are introduced as support for thick clients in multilayered web systems. The challenge is to perform dynamic updating on both local scripts and storage in the web client environment.

Another web-based technology is web services. Essentially, web services are web API (*Application Programming Interface*). They are often part of the cloud and web systems. Similar to modules API, web services provide dedicated functionality to another entity, such as another web service, web or native applications. There is a lack of research on how to apply DSU to web services.

6.4. Applying Autonomous Software Systems Techniques

Besides DSU systems, there are *autonomous* software systems, i.e., adaptive systems, often used in robotics. They share the same primary goals as dynamic software updating systems, availability with changes support. The fundamental difference is that the main logic for providing dynamic updating capability is system built-in, and the whole update process is performed by the programmer [29]. On the contrary, DSU tends to provide programmers transparency, but some DSU approaches are very similar to autonomous systems because they break the transparency as in [36]. Autonomous systems are often based on *control loops* [21] with four steps: *collect* – data gathering from various sources, *analyze* – building a model from collected data analysis, *decide* – making a decision based on the model, *act* – performing decided action. Research of DSU problems related to the point of the update, set of changes, type safety and behavior consistency can be extended with four steps from control loops or similar autonomous decision-making based methods.

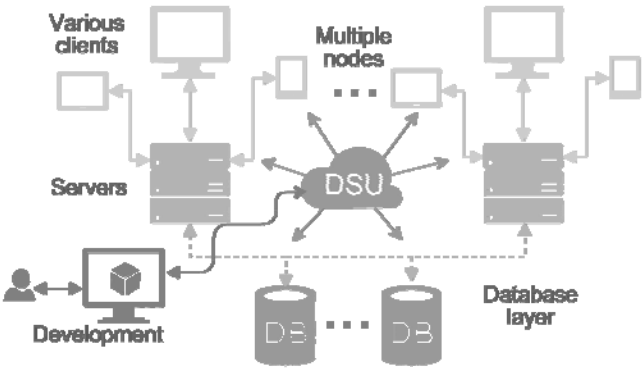

Figure 2. DSU in development and distributed three layer environment

6.5. Runtime Phenomena – State Artifacts

In [12], Gregersen *et al.* discussed *runtime phenomena*, a condition wherein the system state after the dynamic update is applied, is not equal to a system state when the classic update scheme including halt and restart of the system is applied. Following phenomena is detected [12]: *phantom objects* – removed class objects remain in application, *absent state* – object is missing state in newer version, *lost state* – state is lost with type change of the class member, *oblivious update* – features introduced in the newer version are missing, *broken assumptions* – after applying multiple new versions, state and logic dependency assumption may break objects, *transient inconsistency* – application state occurred after the update that will never be reachable by cold restart. More research is needed in both development and runtime environment in order to detect and avoid such conditions.

6.6. Cache Techniques and Complex Structures

Current applications from various real-world domains, such as business and social media, are using large data sets. Every entity in the domain is represented as a data set or abstracted object in memory. Examples are social networks, where users track their daily activities in the form of various multimedia collections with associated metadata and abstract data structures, such as a tree. Although

multimedia data is not often stored in memory, many complex data structures, like entity dependency, are stored in memory in order to achieve a faster and responsive system to the end-users. E.g., user profile activity for smart advertising or end-users information such as "friend" tree for notifications delivery. To increase speed, often accessed data is kept in memory. This mechanism in computer systems is generally called *cache*. Many complex cached structures should be supported by DSU approaches that handle class hierarchy, but they are tested on simple class inheritance examples [11], [37]. In accordance with the cache techniques and complex data structures, the challenge is how dynamic software updating handles state transfer when the associated data structures are changed on the fly.

6.7. Evaluating DSU Implementation

Performance benchmarks from existing research compare execution speed before and after performing a dynamic update. Memory demand comparison influenced by dynamic software updating capability is rarely provided. Such a comparison is needed because current applications are memory demanding. In the case of the DSU concept that supports the coexisting of multiple versions of the running program, the memory demand after multiple updates should be in the worst-case equal to a cumulative number of running versions. On the other hand, a common situation in the computer systems when there is an advantage in the speed of software execution, there is higher memory usage and vice versa. A stable DSU system is required to balance those two requirements in terms of dynamic updating. Further development of appropriate DSU benchmarks, similar to [20], considering the demands for speed and memory resource demands, is needed.

To evaluate availability across the system in different nodes and to check the correct behavior of the system, proper benchmarks are needed. Besides correctness and availability evaluation, the evaluation is also needed for: DSU self-checking analysis, comparing different DSU techniques and methods and their dependability, also as an update failures analysis. The real challenge is how to design and perform such evaluation with different parts of the complex system in mind.

Comparing DSU concept implementation on different platforms increases DSU feasibility. Wernli *et al*. implemented a DSU concept *Theseus* using *contexts* on different programming languages: Smalltalk [36], and Java [35]. Java and Smalltalk are similar as they both use VM (*Virtual Machine*). However, Smalltalk is a fully reflective programming language. DSU concept implementation on different platforms is not usual. DSU concept applied on different platforms, in accordance with the requirements in Section 2, would contribute to the broader use of DSU systems by developers. There is a lack of research on implementation and comparison of DSU concepts on popular object-oriented programming languages, in order to prove concept feasibility.

7. Conclusion

Dynamic software updating has been in the research focus over several decades. Different DSU related problems and approaches to tackle them have been introduced with respect to different software environments, from embedded and operative to distributed and cloud systems. Each environment with specific properties and constraints affects DSU system implementation and architecture. In this paper, we have described the main requirements DSU is trying to fulfill. On the other hand, DSU has specific problems compared to software updating steps: halt, deploy, and restart, leading to the development of various techniques and mechanisms. Although many approaches have been proposed in the literature, outlined challenges indicate the possibility for future work. To cope with challenges, various data analysis on a higher level and decision-making methods can be applied together with improving existing mechanisms and techniques. Further, expanding DSU to the other software environments and platforms could enable new dynamic updating possibilities and approaches.

References

[1]. Bhattacharya, P., & Neamtiu, I. (2010). Dynamic Updates for Web and Cloud Applications. *Proceedings of the 2010 Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, 21–25.

[2]. Cech Previtali, S., & Gross, T. R. (2011). Aspect-based Dynamic Software Updating: A Model and Its Empirical Evaluation. *Proceedings of the Tenth International Conference on Aspect-Oriented Software Development*, 105–116.

[3]. Chen, H., Yu, J., Chen, R., Zang, B., & Yew, P.-C. (2007). POLUS: A POwerful Live Updating System. *Software Engineering, 2007. ICSE 2007. 29th International Conference On*, 271–281.

[4]. Cook, R. P., & Lee, I. (1983). DYMOS: A Dynamic Modification System. *SIGSOFT Softw. Eng. Notes*, *8*(4), 201–202.

[5]. Debugging with the Eclipse Platform [CT316]. (2007, May 1). Retrieved from: http://www.ibm.com/developerworks/library/os-ecbug/ [accessed: 30 December 2019]

[6]. Dmitriev, M. (2001). *Safe evolution of large and long-lived java applications* (Ph.D. Thesis). Department of Computing Science, University of Glasgow, Glasgow G12 8QQ, Scotland.

[7]. Duggan, D. (2001). Type-based Hot Swapping of Running Modules (Extended Abstract). *SIGPLAN Not.*, *36*(10), 62–73.

[8]. Ebraert, P., Schippers, H., Molderez, T., & Janssens, D. (2010). Safely Updating Running Software: Tranquility at the Object Level. *Proceedings of the 7th Workshop on Reflection, AOP and Meta-Data for Software Evolution*, 2:1–2:6.

[9]. Fabry, R. S. (1976). How to design a system in which modules can be changed on the fly. *Proceedings of the 2nd International Conference on Software Engineering*, 470–476. San Francisco, California, USA: IEEE Computer Society Press.

[10]. Frieder, O., & Segal, M. E. (1991). On Dynamically Updating a Computer Program: From Concept to Prototype. *J. Syst. Softw.*, *14*(2), 111–128.

[11]. Gregersen, A. R., & Jørgensen, B. N. (2009). Dynamic Update of Java Applications-Balancing Change Flexibility vs Programming Transparency. *J. Softw. Maint. Evol.*, *21*(2), 81–112.

[12]. Gregersen, A. R., & Jørgensen, B. N. (2011). Run-time Phenomena in Dynamic Software Updating: Causes and Effects. *Proceedings of the 12th International Workshop on Principles of Software Evolution and the 7th Annual ERCIM Workshop on Software Evolution*, 6–15.

[13]. Hashimoto, M. (2007). A Method of Safety Analysis for Runtime Code Update. In M. Okada & I. Satoh (Eds.), *Advances in Computer Science - ASIAN 2006. Secure Software and Related Issues* (pp. 60–74).

[14]. Hayden, C. M., Smith, E. K., Hardisty, E. A., Hicks, M., & Foster, J. S. (2012). Evaluating Dynamic Software Update Safety Using Systematic Testing. *Software Engineering, IEEE Transactions On*, *38*(6), 1340–1354.

[15]. Hicks, M., & Nettles, S. (2005). Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, *27*(6), 1049–1096.

[16]. Kim, D. K., & Tilevich, E. (2008). Overcoming JVM HotSwap Constraints via Binary Rewriting. *Proceedings of the 1st International Workshop on Hot Topics in Software Upgrades*, 5:1–5:5.

[17]. Kramer, J., & Magee, J. (1990). The evolving philosophers problem: dynamic change management. *Software Engineering, IEEE Transactions On*, *16*(11), 1293–1306.

[18]. Lin, D.-Y., & Neamtiu, I. (2009). Collateral Evolution of Applications and Databases. *Proceedings of the Joint International and Annual ERCIM Workshops on Principles of Software Evolution (IWPSE) and Software Evolution (Evol) Workshops*, 31–40.

[19]. Microsoft. (2015). Edit and Continue. Retrieved from: https://msdn.microsoft.com/en-us/library/bcew296c.aspx [accessed: 30 December 2019]

[20]. Mlinaric, D., & Mornar, V. (2017). Dynamic Software Updating in Java: Comparing Concepts and Resource Demands. *Companion to the First International Conference on the Art, Science and Engineering of Programming*. Presented at the New York, NY, USA.

[21]. Nakagawa, H., Ohsuga, A., & Honiden, S. (2012). Towards Dynamic Evolution of Self-Adaptive Systems Based on Dynamic Updating of Control Loops. *Self-Adaptive and Self-Organizing Systems (SASO), 2012 IEEE Sixth International Conference On*, 59–68.

[22]. Neamtiu, I., Bardin, J., Uddin, Md. R., Lin, D.-Y., & Bhattacharya, P. (2013). Improving Cloud Availability with On-the-fly Schema Updates. *Proceedings of the 19th International Conference on Management of Data*, 24–34.

[23]. Neamtiu, I., & Hicks, M. (2009). Safe and Timely Updates to Multi-threaded Programs. *SIGPLAN Not.*, *44*(6), 13–24.

[24]. Neamtiu, I., Hicks, M., Foster, J. S., & Pratikakis, P. (2008). Contextual Effects for Version-consistent Dynamic Software Updating and Safe Concurrent Programming. *SIGPLAN Not.*, *43*(1), 37–49.

[25]. Noubissi, A. C., Iguchi-Cartigny, J., & Lanet, J. (2011). Hot updates for Java based smart cards. *Data Engineering Workshops (ICDEW), 2011 IEEE 27th International Conference On*, 168–173.

[26]. Payer, M., Bluntschli, B., & Gross, T. R. (2013). DynSec: On-the-fly code rewriting and repair. *Presented as Part of the 5th Workshop on Hot Topics in Software Upgrades*, 115–126.

[27]. Pukall, M., Kästner, C., Götz, S., Cazzola, W., & Saake, G. (2009). *Flexible Runtime Program Adaptations in Java - A Comparison* (No. 14). Germany: School of Computer Science, University of Magdeburg.

[28]. Segal, M. E. (1989). *Dynamic Program Updating in a Distributed Computer System*. University of Michigan, Ann Arbor, MI, USA.

[29]. Seifzadeh, H., Abolhassani, H., & Moshkenani, M. S. (2013). A survey of dynamic software updating. *Journal of Software: Evolution and Process*, *25*(5), 535–568.

[30]. Sprenkels, R., & Pras, A. (2001). Service level agreements. *Internet NG D*, *2*, 7.

[31]. Stoyle, G., Hicks, M., Bierman, G., Sewell, P., & Neamtiu, I. (2007). Mutatis Mutandis: Safe and Predictable Dynamic Software Updating. *ACM Trans. Program. Lang. Syst.*, *29*(4).

[32]. Subramanian, S., Hicks, M., & McKinley, K. S. (2009). *Dynamic software updates: a VM-centric approach*.

[33]. TIOBE Index | TIOBE - The Software Quality Company. (n.d.-ah). Retrieved from: https://www.tiobe.com/tiobe-index/ [accessed: 30 December 2019]

[34]. Vandewoude, Y., Ebraert, P., Berbers, Y., & D'Hondt, T. (2007). Tranquility: A Low Disruptive Alternative to Quiescence for Ensuring Safe Dynamic Updates. *Software Engineering, IEEE Transactions On*, *33*(12), 856–868.

[35]. Wernli, E. (2012). Theseus: Whole Updates of Java Server Applications. *Proceedings of the 4th International Workshop on Hot Topics in Software Upgrades*, 41–45.

[36]. Wernli, E., Lungu, M., & Nierstrasz, O. (2012). Incremental Dynamic Updates with First-Class Contexts. In CarloA. Furia & S. Nanz (Eds.), *Objects, Models, Components, Patterns* (pp. 304–319).

[37]. Würthinger, T., Wimmer, C., & Stadler, L. (2013). Unrestricted and safe dynamic code evolution for Java. *Science of Computer Programming*, *78*(5), 481–498.