Prepare to Die: A New Paradigm for Live Update

Cristiano Giuffrida
Department of Computer Science
Vrije Universiteit, Amsterdam
c.giuffrida@few.vu.nl

Andrew S. Tanenbaum
Department of Computer Science
Vrije Universiteit, Amsterdam
ast@cs.vu.nl

Abstract

Many real-world systems require continuous operation. Downtime is ill-affordable and scheduling maintenance for regular software updates is a tremendous challenge for system administrators. Existing live update approaches proposed as a solution to this problem have failed to reach broad acceptance in system administration communities.

In this paper, we investigate the root causes of the poor acceptance and argue that a new model is necessary to offer adequate dependability guarantees. After describing the new model, we propose a taxonomy of live updates and analyze many practical examples from operating systems. We show how the nature of the update is crucial to determine the properties and limitations of the resulting live update process and discuss the emerging need for update-aware systems.

Keywords

Live Update, Online Maintenance, Configuration Management, Availability

1 Introduction

The past decades have witnessed an increasing demand for highly reliable computer systems. The need for continuous operation is emerging in many areas of application with different levels of impact. Mass market software systems, not initially conceived with extreme availability in mind, attract more and more consumers who expect nonstop operation. Many unsophisticated users find it very annoying to reboot their PC after an update or a crash. For workstation users in companies, reduced availability directly translates to productivity loss.

In industrial systems, the need for continuous operation is even more evident. In many cases, high availability is required by design. As an example, the telephone network, a 99.999% availability system, can tolerate at most five minutes of downtime per year [1]. In other applications, such as factories and power plants, availability constraints are even more tight. In general, downtime constitutes a serious threat to high-availability systems, as it results in loss of transient state and disruption of service.

In business systems, absence of service leads to revenue loss. It has been estimated that the cost of an hour of downtime can be as high as hundreds of thousands of dollars for e-commerce service providers like Amazon and eBay and millions of dollars for brokerages and credit card companies [2]. In addition, long-lasting periods of downtime for popular services are newsworthy and affect user loyalty and investor confidence. When eBay suffered a 22-hour site-wide outage in 1999, the significant impact on the public image of the company caused 26% decline in stock price and an estimated revenue loss between $3 million and $5 million [3].

In mission-critical and safety-critical systems, downtime or unexpected behavior can lead to catastrophic consequences. For example, unplanned downtime in a widely deployed energy management system caused a blackout affecting 50 million people in U.S. and Canada in August 2004 [4]. Another famous episode relates to the Patriot missile defense system used during the Gulf War. A software flaw prevented the interception of an incoming Scud missile, leading to the death of 28 American soldiers [5]. Many other examples of mission-critical systems can be found in aerospace, air control, telecommunication, public information, transportation, industrial control, military, and medical applications.

Unfortunately, software changes over time. Despite decades of research and advances in technology and software engineering, the majority of cost and effort spent during software lifetime still goes to maintenance [6]. System administrators have learned to live with the many changes that programs undergo over time. The introduction of new features, enhancements, bug fixes and se-

curity patches are the norm rather then the exception in widely adopted software solutions.

Current trends in software development suggest that this tendency will likely grow in the future. The complexity of software systems is increasing dramatically, and so are the number of bugs, security vulnerabilities and unanticipated changes. Studies have determined that the number of bugs in commercial software ranges from 1 bug per 1000 lines of code to 20 bugs per 1000 lines of code [7]. As a result, software vendors continuously release new updates and publish recommendations.

The increasing number of updates available represents an enormous burden for system administrators. Whether it is a small patch or an entire new version of the system, an update requires operations like copying, compiling, or running particular installation procedures. At the end, a system reboot is conventionally required before the changes can take effect.

Rebooting constitutes a major problem for the management of systems that must provide strong availability guarantees. In addition, in some areas of application—physical systems for example power plants—shutting down and restarting the system may take a long time. Not surprisingly, previous studies have indicated that maintenance is the primary cause of downtime for high-availability systems, possibly accounting for as high as 75% of the outages [8].

Fortunately, most high-availability installations are replicated across multiple machines. To mitigate the effects of downtime, system administrators usually update one node at time when a new patch or version becomes available. This approach, however, has shortcomings for both small and large installations.

In small installations, shutting down a single node leads to a capacity problem. The machine must be made unavailable for the entire duration of the update process, and other nodes may get overloaded from the increased number of service requests. Oversizing is a common way to address this problem, but additional cost is required.

In large installations, an incremental update process may produce undesirable effects. When a new feature is included in an update, it may take a long time before all the nodes are updated. As a consequence, while the update is still in progress, repeated service requests that happen to land on different nodes may lead to different results. In addition to the inconvenience caused to the users, increased application complexity is often necessary to avoid possible inconsistencies.

In practice, planned downtime to update a software system is hard to schedule whenever high availability is of concern. As a result, system administrators tend to forego installing updates or decide not to install them at all [9]. This, in turn, increases the risk of unplanned downtime caused by software bugs or security attacks and can result in even more catastrophic failures. As a matter of fact, it has been estimated that more than 90% of security attacks exploit known vulnerabilities [10].

1.1 Live Update

Experience indicates that trading off high availability against the need to update a software system is painful for many applications. Live update—namely the ability to update software without service interruption—is a promising direction to address this problem. An infrastructure to apply online changes to a running system would greatly aid in the maintenance of systems that cannot tolerate disruption of service or loss of transient state. Live update could be the definitive solution to support software evolution in high-availability environments.

In the near future, the importance of live update will likely grow and its application will not be limited to high-availability systems. With the advent of ubiquitous computing, software tailored to ordinary devices will become more complex, thus incurring frequent maintenance updates. Emerging embedded software will necessarily require live update approaches to maintain the current level of transparency and be widely adopted. It is very unlikely that most people will understand the need to turn off their car as a consequence of a high-priority security update. Even worse, one could envision legal consequences when a user is forced to update and restart his TV set while watching a pay-per-view sporting event.

Live updatable software systems have also other potential benefits [11]. First off, live update offers a higher degree of flexibility with respect to when to start the update process. Different update policies can be employed to start the live update process at an appropriate time. A low-priority update policy could be labeled *update when the system load is low* in an attempt to minimize system disruption, whereas a critical security update might be labeled *update as soon as possible*.

In addition, live updates can be used for fast prototyping. When developing a new version of the system, the ability to test changes online may save the programmer a lot of time. This is especially true for systems that take a long time to reboot.

Finally, live updates can be used to dynamically address the behavior of a running system. Applications include: (i) efficient dynamic monitoring—a monitoring infrastructure can be loaded and unloaded dynamically when needed with no runtime overhead in the normal case, (ii) specialized support and optimizations—customizations tailored to specific scenarios can be added dynamically when required, (iii) adaptive algorithms—system-triggered live updates can be employed to continuously adapt the behavior of the system to the monitored workload.

1.2 Contributions of this Paper

The contributions of this paper are threefold. First, we discuss different models for live update and analyze characteristics of state-of-the-art solutions described in the literature. Our aim is to investigate the root causes of why live update technologies have failed to receive broad acceptance.

Second, we propose a new model for live update targeted towards building dependable and trustworthy solutions. Our concrete goal is promoting the design of systems and infrastructures specifically conceived with live update in mind, as opposed to building live update suites targeting transparency and backward compatibility.

Finally, we propose a taxonomy of live updates focusing on the nature of the update rather than the state of the system when the update arrives. The taxonomy aims to provide criteria and scenarios to establish and analyze the level of complexity and potential disruption of a given live update. We develop our analysis through concrete examples and investigate the properties and limitations of live update.

We believe our contribution is an important step to improve the common understanding and perception of live updates and identify the issues to design dynamically updatable systems. The ultimate goal is to foster the development of systems that support dependable live update by design and encourage system administrators to use them. To the best of our knowledge, no one before has provided a comprehensive analysis of the properties and limitations connected with live update, established an adequate taxonomy based on the aforementioned criteria, and investigated system design issues from a broad perspective.

1.3 Roadmap

The remainder of the paper is laid out as follows. We first discuss different models for live update highlighting characteristics, limitations, and trade-offs (Sec. 2). Then we present the taxonomy (Sec. 3), analyze its implications, and discuss our findings (Sec. 4). Finally, we survey related work (Sec. 5) and conclude with final remarks (Sec. 6).

2 Models for Live Update

Live update has received a great deal of attention in the past two decades. Both hardware-based and software-based approaches have been extensively studied in the literature.

Hardware-based solutions rely on specialized hardware and adopt a primary-backup scheme to update without service interruption. The main shortcoming of this approach is the additional cost for redundant hardware and the introduction of synchronization complexity to preserve transient state and avoid inconsistencies. Despite their drawbacks, these solutions have become popular in many large organizations. Dealing with an average of 20,000 updates per year and yet maintaining more than 99.5% availability, Visa's transaction processing system is a notable example in this category [12].

In contrast, software-based solutions provide runtime support to apply online changes to running software. These approaches can be further categorized basing on the structural unit of change they support. Live update frameworks described in the literature usually allow dynamic replacement of functions, objects or processes. A number of techniques at each level of granularity have been proposed in several research communities. More recently, system-level approaches to update the system as a whole by running two parallel instances [13] or using virtualization technologies [8, 14] have also been explored. These approaches, however, require ad-hoc infrastructures—a separate execution environment or virtual machines—and are usually more tailored to major system updates.

Another possible classification criterion for software-based solutions is the type of software supported. Some approaches target software applications in general, with possible restrictions on the environment or programming language used. Many other solutions are tailored to updating specific operating systems.

In the present paper, we mainly focus on software-based live update for operating systems. Nevertheless, the same considerations apply to other software systems sharing a similar event-driven model. In an operating system, an event is primarily determined by an application requesting service by means of a system call or by a hardware interrupt. In server applications, such as HTTP servers or database management systems, events are triggered by user requests and processed in bounded time. The analysis presented in this paper is equally relevant to both scenarios.

There are at least three good reasons to concentrate our attention on operating systems. First, operating systems are plagued with continuous maintenance updates. As a consequence of their size and complexity characteristics, modern operating systems have probably more bugs and security holes than any other piece of software [15]. In addition, their code base rapidly evolves to support new devices and applications.

Second, high availability of operating systems is a major concern. Downtime in an operating system directly translates to downtime for all the hosted applications. Moreover, recovery time is substantial. After a conventional software update to the operating system, a machine reboot is required to restart the operating system. As a

result, minimizing downtime due to maintenance is extremely challenging.

Finally, operating systems offer a complete set of functionalities and update scenarios. This property is useful for our analysis. Furthermore, system administrators, programmers, and other categories of users are already familiar with many real-life examples of operating system updates.

2.1 Existing Approaches to Live Update

We now turn our attention to live update solutions described in the literature. System research in the area of live update is generally focused on designing frameworks to seamlessly apply online changes of any sort to existing software systems the instant they arrive.

The dominant approach is to glue changes into the running system by loading the update, executing a state transfer function provided by the author of the update, and redirecting execution to the new version. Loading the update is usually accomplished through some form of dynamic linking or special support offered by the runtime environment—linking a component in a component-based system, for example. The purpose of the state transfer function is to convert the old state of the system into a valid new state before resuming execution in a consistent way. For example, if new data structures are included in the update, the state transfer function must be executed to initialize them with meaningful values before the new version can start executing properly. When state transfer completes, execution is redirected to the new version by exploiting some form of indirection mechanism specific to the language or runtime environment used. Many techniques to add a level of indirection and redirect execution are described in the literature at different levels of abstraction, such as: function pointers[16], dynamic instrumentation techniques[17], indirection tables[18], interceptors and naming services [19]. The live update framework incorporates the ability to compare and analyze the old version and the new version of the system to figure out how to apply changes correctly and resume execution in a consistent way. Using these techniques, execution is resumed on the new version seamlessly, without the original version being aware of the update.

As a result, much attention has been dedicated to performance, backward compatibility and transparency. These properties have been largely promoted as key success criteria for live update technologies.

Performance measurements have been extensively used to determine the time and the overhead required to apply an online change. In addition, performance statistics have been gathered to measure the overhead on the runtime behavior of the system. This metric is relevant to estimate the level of intrusiveness of a live update framework.

Backward compatibility measures the ability of a live update framework to support and integrate into existing software systems. An approach designed to retain binary compatibility can support precompiled software even if the original source code is not available and the software was not designed with updating in mind. In contrast, an approach targeting source compatibility aim to provide a toolchain to build a live updatable version of the original software.

Finally, transparency relates to the property of hiding the details of the live update process from authors of the update, system administrators, and the system itself. This property is usually regarded as the ability of the framework to support existing binary or source patches and dynamically apply them without any further modification or user intervention. The only exception is normally the state transfer function that must be manually provided when the new version uses different data structures than the old one.

Live update solutions with emphasis on these properties have been studied for years now. Despite significant effort, few research systems have made their way into the real world. The majority of high-availability systems still rely on custom solutions or do not use live update at all [20]. Most approaches have failed to reach broad acceptance because dynamically updatable systems as such are not yet considered a trustworthy solution. We believe that the common perception reflects the lack of practicality and neglect of important dependability properties in existing live update solutions.

First off, research in being able to apply an update the very instant it is released is dubious, given the long time it usually takes to find and fix the bug and publish the patch. Studies of operating systems have estimated an average bug lifetime of about 1.8 years, with the median around 1.25 years [15]. Even once a bug is discovered, it may take days or weeks to resolve it and deliver an adequate patch. In addition, experience indicates that deciding when to apply the update is crucial to avoid problems induced by possible bugs in the patch itself. Previous work on security patches suggested that system administrators should delay an update at least 10 days after the patch's release [21]. In other words, let somebody else be the guinea pig.

In this light, delaying a live update for a few seconds to allow the system to get into a known state seems to be a reasonable option. If the bug has been there for well over a year, delaying the fix for another 5 seconds is unlikely to be fatal, especially if you have already intentionally waited 10 days to see if anyone reports bugs in the update. Yet most of the research in the area assumes the update must be applied the instant it is available. In

practice, deferring the update process until specific constraints are met by the running system is desirable for several reasons. First, applying online changes when the system is in a known, stable state is likely to result in a more reliable update process. In addition, the ability to specify policies that determine when to schedule the process offers a higher degree of flexibility and predictability guarantees. This aspect has largely been neglected in the literature.

As far as backward compatibility and transparency are concerned, the focus in previous work is largely motivated by the need to cope with existing systems. Although its importance in supporting legacy systems is indisputable, we argue that a model based on these properties poorly fits in a standard software development process.

Existing solutions assume that the live update infrastructure is invisible to the development process leading to the construction of an update. This, in turn, delegates to the infrastructure the responsibility of applying the update and ensuring that the resulting configuration is valid. Unfortunately, inspecting a patch or a new version of the system to figure out what changes occurred and how to apply them correctly in the general case is complicated and error-prone, probably closer to reverse-engineering than system design. Conversely, programmers developing a new version of the system are certainly aware of all the changes that they made and can provide directions on how to apply them properly at runtime.

Unfortunately, the emphasis on backward compatibility and transparency in the literature has fostered a clear separation between the development of a system and the ability to apply online changes to its running instances. The main focus is on designing live update infrastructures that can transparently glue into a running system any patch at any time. As a result, many live update solutions are primarily concerned with complexity and type safety, as opposed to ensuring the general safety of an update [22].

In the literature, the role of safety constraints in live update solutions is controversial. The reason for this lies probably at the theoretical foundations of live update. Pioneering work on the validity of a live update was undertaken by Gupta [23] and has been highly influential in succeeding research.

In Gupta's work the validity of a live update is formally proven undecidable in the general case. Namely, given an arbitrary system at an arbitrary point in time, an online change, and a state transfer function, it is not possible to determine if the update will result in a valid configuration for the system.

Gupta's result has led many researchers to neglect update safety and focus more on type safety and other properties. Many models impose restrictions on the type of an update, others ignore update safety or conservatively assume that system maintainers can somehow be given the responsibility to recognize whether or not an update is valid. Even models that do not target backward compatibility but aim to provide frameworks to create live updatable systems have largely ignored update safety.

Other studies have tried to establish strong update safety conditions tailored to specific system models. Common definitions used in component-based systems are *passivity*, to indicate the state of a component not processing any request, and *quiescence*, to indicate the state of a passive component whose directly or indirectly dependent components are all passive [24]—note that in other research communities quiescence refers to inactive code in general [17]. For example, consider a component that performs logging in a transaction-processing system. If no logging is in progress, the component is said to be passive. If not transaction that involves logging is in progress, the component is also said to be quiescent.

These definitions have been largely acclaimed as a stable component state to perform a valid dynamic replacement. Unfortunately, assumptions on the nature of the system somehow limit the applicability of these results. In addition, experience suggests that using a single general-purpose condition to establish the validity of a live update reduces the degree of flexibility and causes excessive system disruption in the average case [25].

2.2 Our Approach to Live Update

We envision a system that can support live update by design. In our model, the *nature* of an update is central. We believe published updates should contain more information about what they affect and how, thus allowing the system to determine when to apply them safely. Note that we are talking about both small security patches as well as functional changes from one version to the next.

We believe that a paradigm shift is necessary to realize this vision, moving from the common belief that live update is somehow similar in spirit to conventional patch installations. As a matter of fact, there is at least one fundamental difference. When changes are applied online, the impact on the system and the validity of the resulting configuration depend on the nature of the update and the state of the system at the exact moment the update is performed. A bad update timing can cause serious consequences. To address this challenge, we believe that a tighter integration between live update and software development process is necessary. Both the programmers and the system should be aware of changes and be prepared to deal with them.

In particular, the programmers producing the update should document the changes they made in a *live update*

package. Note that we use the new terminology to indicate the distinction between patch installation and live update. A live update package contains a patch or a new version of the system, but also information about the nature of the changes and directives on how to apply them online properly.

Software, in turn, should be designed with live update in mind and be prepared to cooperate in the (inevitable) update process. We need to find structural ways to deal with that. In our model, the system supports an infrastructure to interpret programmers' specifications and apply the changes correctly at an appropriate time. When an update becomes available, the system allows an update manager to send all components affected by the update a "Prepare to die" message that gives them directives on how to terminate properly. Typically, a component will first be asked to finish the necessary pending activities to ensure that changes are applied only when its internal state is stable. Then, it will save its state in a safe place (in another component's address space or on a disk) in a predetermined, known, format. Finally, it will send back to the update manager a "Ready to die" message. When all components have responded, the system is ready to be updated. In spirit, this design is similar to a two-phase commit. Then the new components are loaded into the running system. At that point, state transfer takes place and the new version can safely resume execution.

We believe that this paradigm—having the software actively cooperate in the update—is far easier and more practical than the reigning paradigm in the literature (transparency) in which the update is an unexpected bolt out of the blue. Developers know that updates happen and can prepare the code to deal with them.

Our approach results in higher flexibility than using a general-purpose safety condition and better safety guarantees than starting the update process at an arbitrary moment. In particular, the proposed model solves the problem of establishing a safe update time structurally, using a deterministic live update process. By design, the system can recognize the nature of each update and, in response to an update request, is able to reach a stable, receptive state before applying changes. The appropriate stable state to perform an update is determined from the nature of the update itself.

A higher degree of flexibility can also enable the definition of custom update policies to automatically start the update process at a point in time when eventual disruption caused by a crash could be minimized. This is an important aspect to improve the predictability properties of live updates. Even assuming a safe live update process, the resulting configuration can in fact still be unstable if the update itself contains bugs. Moreover, some classes of live updates that take a long time could be scheduled when the system is lightly loaded to reduce disruption. Finally, whenever the system cannot meet required constraints or has already entered a tainted state due to latent errors, the live update process should not occur at all unless an appropriate recovery mechanism is available.

In addition, our approach offers better support to the programmers creating the live update package. In models assuming the state of the system is not known in advance, designing a live update package or custom code to execute at update time—the state transfer function or other initialization code—can become overly complicated and error-prone. A programming model that forces programmers to deal with an arbitrary number of possible initial states is unlikely to be effective and reliable. Programmers are instead familiar with a deterministic model, where the initial state of execution is unique and well known. We envision a similar programming model for live update.

Furthermore, our model simplifies testing of live update packages. In existing solutions, testing is largely ineffective in verifying the correctness of an update. Since the update can be applied at an arbitrary moment, changes should theoretically be tested against all the possible system states. This is clearly infeasible. As a result, live updates rarely go through extensive testing, making the final update process even less reliable. Our approach, in contrast, makes testing a deterministic and effective process.

Finally, we remark that our model does not target binary compatibility and is by no means transparent. We regard these concepts as part of the problem, not part of the solution. There is no real need for instant update and it is very hard to do it right, so it should not even be considered when reliability is of concern. Delaying for at most a few seconds until the system has reached a predictable state is not harmful and much simpler. In addition, we believe that our approach can perfectly fit in a typical software development process, also promoting a better system design and encouraging programmers to document changes.

3 The Taxonomy

In the previous section, we argued that the nature of the update is a crucial aspect to determine the properties of a live update process and the impact on a running system. In this section we develop this intuition further and propose a taxonomy of live updates.

Rather than focusing on formal definitions, we propose criteria to describe the nature of an update and discuss possible scenarios through examples as this will be of more use to most system administrators. Each scenario in the taxonomy defines a category of updates with an increasing level of severity, resulting in higher update complexity and more disruptive effects for the system.

Before detailing our analysis, it is appropriate to introduce the reference model used. In the following, we refer to a UNIX-like operating system with a standard interface and a generic live update infrastructure. The resulting software system is composed of a number of dynamically updatable structural units. Depending on the architecture of the operating system and the runtime support provided by the live update infrastructure, the structural unit used may be a function, object, or process.

We model the interactions between structural units by means of generic message-passing. A message can be interpreted as a function call for a function, a method invocation for an object, and a signal or IPC call for a process. The execution of the code of a structural unit follows upon reception of a message. As in a standard event-driven model, the main message flow is generated in response to a system-level event.

The semantics of an interaction between multiple structural units is defined by a protocol. We model a protocol as the sequence of messages exchanged between an initiator and one or more structural units that act as recipients. The initiator is the structural unit that starts the protocol in response to a message received that requires further processing.

Given the definition of structural units and their interactions, we propose the following criteria to describe the nature of an update.

Changes to code.
Changes to code refer to changes to algorithms or protocols and affect one or more structural units.

Changes to data.
Changes to data refer to changes to data structures used by one or more structural units.

Resource-sensitive changes.
Resource-sensitive changes refer to changes that impose new requirements for fundamental resources upon which the operating system relies. In our analysis, we primarily refer to hardware resources. Examples include memory, disk, and peripheral devices.

3.1 Definition

Following the characterization of changes introduced earlier, we present the taxonomy of live updates broken down into six categories.

1. Update affects one structural unit
This category comprises changes to data and algorithms isolated in a single structural unit. Common updates in this category are small bug fixes, security patches, and performance improvements. An example of a bug fix is changing a test for $i < j$ to $i \leq j$. An example of security patch is performing length checking on an input string to avoid buffer overflow attacks. An example of performance improvement update is a new algorithm that first checks for the common case before using a more general and slower approach.

2. Update affects protocol
This category comprises changes to a protocol between two or more structural units and may include changes to code and data. Changes to a protocol refer to changes to the number or type of recipients, changes to the number, order, or semantics of the messages exchanged, as well as changes to the content or meaning of any of the fields in a message. An example is changing the message format for a call to the disk driver to represent a block number in 48 bits instead of 32. Another example is a change that adds, changes or removes a message to trigger error logging in the structural unit that loads kernel modules.

3. Update affects global data
This category comprises changes to global data structures that are shared across multiple structural units. Also in this category is a change to global data constants, like renumbering all the error codes. Other examples are changing the internal representation of a process identifier from 16 bits to 32 bits or of an inode shared across multiple structural units throughout the system. An additional example may be a change to data shared in a specific subsystem, such as a change to the format of internal IOCTL codes.

4. Update affects global algorithm
This category comprises changes to a global algorithm that may affect multiple structural units. An example in this category is moving the code to add a new inode to the inode table to a different structural unit, as a consequence of system restructuring. Another example is an improved implementation of a file usage counter. Assume the original version incremented a counter in the inode at $open()$ time. Imagine that, after noticing that some files are opened but never accessed, the code to increment the counter is moved to the time when the first $read()$ or $write()$ is processed.

5. Update affects data on the disk
Updates in this category are generally concerned with data stored on the disk. A first example is a change to the format of the disk image used for process checkpointing. Another example is a change to the encoding of temporary files for internal use. More advanced examples include: (i) changing the executable format, or (ii) changing the file system format, for example to store

additional information (e.g. more disk addressed) in the inode on the disk.

6. Update affects hardware requirements
This category comprises changes that impose new hardware requirements. Examples include changes to minimum requirements for storage, memory, or processor speed and changes to hardware supported. Practical examples in this category can be found in many new releases of publicly available operating systems. For example, with the release of Mac OS X v10.5 (Leopard), Apple dropped support for all PowerPC G3 processors and for PowerPC G4 processors with clock speeds below 867 MHz. Another example is the transition from Windows XP to Windows Vista. Minimum requirements went from 64 MB to 512 MB for RAM and from 1.5 GB to 15 GB for disk space available. In addition, Vista dropped support for older motherboard technologies like the ISA bus and APM and for every graphics card not compatible with the DirectX 9 specifications.

3.2 Consequences

In this section, we discuss each category of live updates in detail and analyze the consequences for the update process. The gold standard is being able to do with live update something that previously required a reboot. As we will show, this is not always possible, but we would like to get as close as we can.

3.2.1 Update affects one structural unit

In the simple case, the update can be performed by atomically replacing the structural unit. That is, we can apply changes when the structural unit is not processing a message. Recall the security patch example proposed earlier. If we replace the structural unit when no message is being processed, all the messages following the update will use the new code and be verified as expected to avoid possible buffer overflows. The same considerations apply to the bug fix example, but state transfer is necessary to initialize the new data type correctly.

In other cases, an update that uses atomicity at the structural unit level may not be as effective. For example, imagine a protocol to write a chunk of data to a file. The protocol consists of multiple iterations between the virtual file system layer and a specific file system implementation. Assume that the original file system implementation used buffered writes and only flushed all the content received at the last interaction. If the file system implementation is changed to perform unbuffered writes, the change affects only a single structural unit. Yet, if we allow the replacement of the file system when the protocol is in progress, additional state transfer is necessary to flush the content of the buffer to the disk before resuming execution. If the update used atomicity at the protocol level—that is changes are applied only when the protocol is not in progress, no state transfer would be necessary.

In more advanced cases, atomicity at the structural unit level may be insufficient to apply online changes correctly. Consider the same protocol described above. Assume that the file system implementation is changed to collect statistics on the duration of a $write()$, storing a timestamp when the first message from the virtual file system layer is received and another one at the last interaction. If changes are applied when the protocol is in progress, no state transfer is possible to bring the new version to a valid state. Atomicity at the protocol level would make the update feasible and simple. As an alternative, if some imprecision is tolerable in the statistics collected, the state transfer function can be instructed to use the timestamp of the time changes are applied.

3.2.2 Update affects protocol

In the simple case, the update can be performed by replacing all the structural units affected when the protocol is not in progress. Recall the driver operation example. In this scenario, the message format used in the protocol is changed. If we replace the driver and the counterpart when there is no communication in progress, all the following protocol instances will use the new format without breaking the semantics of the protocol.

In other cases, the update may require synchronization with additional structural units. For example, imagine a filter driver that detects low-level data corruption. The driver intercepts each write request to the disk driver and breaks it down into a first call to write the data block to the disk and a subsequent call to read the content back and compare it with the original data block. Consider an internal module of the filter driver that compares the two blocks. Assume the module is a structural unit that exposes a service protocol to receive the original block in the first message and the block read from the disk in a second message. To implement the service efficiently, a single-message inner protocol is used to interact with another structural unit whose job is computing the checksum for each block received in the message. If in a new version of the system the inner protocol is changed to use a more efficient checksumming algorithm, changes also affect the execution of the service protocol. If we replace the module and the checksum helper by using atomicity at the inner protocol level, no state transfer is possible to bring the new version of the module to a valid state in the general case, because the original data block may have been lost. In contrast, if we allowed the update at a time when neither the inner protocol nor the service protocol were in progress, the resulting configuration would

be valid and no state transfer necessary.

3.2.3 Update affects global data

In the simple case, the update can be performed by replacing all the structural units affected when none of them is actively accessing the global data changed. Recall the process identifier example. Assume we changed the internal representation of the process identifier to use a larger data type. If the identifier is shared, for example, between two separate structural units such as the process manager and the memory manager, the update can be performed when both structural units are not actively processing a message that involves access to the identifier.

In other cases, the update may require higher levels of synchronization. Recall the error code example. Assume we introduced additional internal error codes for an $exec()$ system call to handle unexpected error conditions with a finer level of granularity, for example. If we allow the replacement of all the structural units affected when the system call is in progress, the resulting configuration may not behave correctly. In particular, some of the new error conditions may not have been recorded in the old version of the code before the update was performed. In that case, no state transfer is possible to bring the new version to a valid state. In contrast, if we allowed the update only at a time when the system call was not in progress, the resulting configuration would be valid and no state transfer necessary.

3.2.4 Update affects global algorithm

In the simple case, updates in this category require proper synchronization between all the structural units affected. Recall the file usage counter example, where an update moves the code to increment a file usage counter from $open()$ to the first time $read()$ or $write()$ is processed. If the update is performed when no file is opened, the resulting configuration is valid and no state transfer is necessary. In the opposite situation, state transfer is required to adjust the value of the counter properly. In particular, for each open file, the state transfer function should decrement the counter if the file has never been read or written before. How hard it is to access this information determines the level of complexity of the state transfer function. If this information is not accessible, no state transfer is possible to bring the new version to a valid state.

In other more advanced cases, live update may not be possible at all. For example, consider a change to the generation algorithm of the random number generator. If running applications or structural units of the operating system rely on a sequence of random numbers provided by the generator, a live update would break this assumption regardless of when changes are applied. The only reliable solution here is a conventional reboot update.

3.2.5 Update affects data on the disk

In the simple case, the update can be performed by replacing all the structural units affected when none of them is actively accessing the changed data. Recall the process checkpointing example and consider an update to support a compressed disk image. Assume the disk image is shared between two structural units to respectively checkpoint and resume execution of a process. When the structural units are not actively processing a message, the update can be safely performed. A state transfer function will be necessary to read the content of the image from the disk, compress existing data, and write everything back to the disk. The duration of the update process and the impact on the system depend on the size of the disk image and the complexity of the compression algorithm.

In other cases, a reboot may be desirable or required to update the system. For instance, imagine that the file system format is changed. Assume that the format of the inode on the disk is changed to support 32-bit UIDs. If a spare partition is available, the update can be performed live although slowly. The system can run *mkfs* on the new partition, laying down the file system in the new format and then copying all the files. When they are all copied, it has to go back to copy files changed since copying began, repeatedly until done.

In more advanced cases, the update on an existing system may not be possible at all. Consider a change in the file system format to count the number of times every file has been accessed since creation. No state transfer can bring the new version of the system to a valid state. But neither can a reboot. It cannot be done at all.

3.2.6 Update affects hardware requirements

Updates in this category can only be supported if existing hardware matches the new requirements. Consider an update that changes the minimum RAM requirements from 512 MB to 1 GB. If the machine has already 1 GB of RAM available, the update can be applied immediately. If it has only 512 MB, new hardware (more memory) will have to be purchased and a reboot done.

4 Discussion

In the previous section, we analyzed the consequences of several scenarios drawn from the categories proposed in the taxonomy. Our analysis did not aim at generality but was instead driven by concrete examples to explore the

properties and limitations of live update. Empirical evidence has supported our original intuition. Each scenario revealed an increasing level of severity of an update from different perspectives. In the following, we discuss our findings.

First, a live update is not always feasible. We showed examples where no synchronization mechanism and state transfer function could be provided to perform a live update resulting in a valid configuration for the system. In many of those cases, a reboot is necessary to perform the update. In other cases, manual intervention of the system administrator may be required. In the most unfortunate cases, the update cannot be done at all.

Second, a live update is not necessarily desirable. In some cases, the live update process can cause significant disruption for the running system. As the complexity of changes and state transfer increases, the update process may take longer and the impact on the system become more evident. In particular, a resource-consuming update process may be problematic or not feasible at all if, for example, state transfer involves copying large chunks of memory and not enough extra memory is available. When substantial disruption is expected, applying changes online can be inconvenient.

Third, the constraints required for the system at update time vary. We observed that updates of different natures may require different levels of atomicity to be applied online. In simple cases, no synchronization is necessary to perform the update. In other cases, atomicity at different levels may be required to guarantee a safe update process and a valid resulting configuration for the system. We also noted that, for higher levels of severity, enforcing the level of atomicity required is increasingly difficult and expensive.

Finally, the complexity of state transfer depends on the constraints imposed at update time. In many cases, we observed that the level of atomicity required at update time can be relaxed. Nevertheless, as we gradually relax constraints imposed at update time, we observe an increasingly complicated state transfer. In some circumstances, constraints cannot be further relaxed or state transfer will become infeasible. Following these considerations, we recognize the need for a more general definition of state transfer that also considers the constraints imposed on the system at update time. Trading off the implementation complexity of the state transfer function against the number of constraints to impose at update time will probably be an important design decision for programmers of systems that support live update by design. The nature of the update and the context will drive the final decision.

In summary, important results can be drawn from the scenarios presented. For high levels of severity, live update—or even a conventional reboot update for that matter—may be expensive or infeasible. But in most other cases, the properties of the live update process are well-defined. Given an update with known characteristics, a desirable stable state for the system at update time can be established if affected components of the running system cooperate when they are told "Prepare to die". The target state depends on the nature of the update and the constraints imposed by the programmers for the state transfer function.

In addition, our investigation shows that the dominant assumptions used in the literature may lead to undesirable effects. In particular, restricting the design to a unique stable state at update time will result in reduced flexibility with important consequences. For example, using quiescence [24] as the only stability condition is unnecessarily expensive in the average case. In highly connected systems such as operating systems, this condition translates to synchronizing a large part of the system regardless of the nature of the update. As a result, it may be necessary to freeze the entire system even to apply a minor and local bug fix. Furthermore, for updates with high levels of severity this condition may not even provide adequate support. For instance, recall the file usage counter example. If we want to avoid updating when applications have still some files opened, blocking the entire system will not really be of any help.

In the opposite direction, assuming that updates can be performed at an arbitrary moment (transparency) results in poor stability guarantees for the update process. As a result, the complexity of state transfer grows unnecessarily with increasing levels of severity, forcing the programmer to deal with more and more undesirable conditions. Imagine changing the semantics of a protocol between the process manager and the virtual file system layer and allowing the update while the protocol is in progress. The complexity of state transfer would reflect the complexity of the protocol and the changes made. In addition, some examples revealed that this assumption restricts the number of updates that is feasible to perform online without compromising the overall validity of execution. A live update solution that combines this model with transparency is likely to incur the safety problems discussed earlier in the paper.

Dealing with indeterminacy is neither desirable nor necessary. No one would approach concurrent programming with no adequate language support. We believe adequate system support is equally crucial for live update. The system should be able to support a number of stable states for the update process. Depending on the nature of the update, it should be possible to determine the appropriate stable state for a safe and predictable update process. This principle faithfully reflects our approach discussed earlier.

To conclude our analysis, we comment on the levels

of granularity examined. If system support is to be integrated in a live update solution, we expect the structural unit of change to play an important role in the design. Final conclusions cannot be drawn, but it is clear that higher levels of granularity are more appealing as the complexity and potential evolution of the system increases. For example, if several function signature changes are to be expected in each update, supporting live update at the function level is probably not a good idea. For complex systems that tend to go through a lot of changes, higher levels of granularity represent a better option.

5 Related Work

To our knowledge, no previous study has tried to assess the general properties and limitations of live updates from a broad perspective and establish an adequate taxonomy based on the nature of an update. Classifications of update types from a functional point of view have been occasionally proposed to illustrate the properties of a live update solution [17].

As for update safety and other dependability properties, previous work is largely concerned with theoretical aspects and standard definitions for the validity of an update in general.

Gupta [23] and other researchers [26] deal with the general undecidability of the validity of an update and formalize sufficient conditions in specific application domains. The focus here is on formal definitions rather than system design.

Bloom and Day [27] investigate the limitations of state transfer in the general case when the original specifications of a module are violated. Our analysis generalizes the state transfer problem and shows how, given an update of a particular nature, the feasibility and complexity of state transfer vary depending on the state of the system at update time.

Kramer and Magee [24] describe a model for distributed systems and propose the use of transactions to ensure atomicity. The general validity of an update is determined by ensuring that each event-generated transaction is entirely executed on a single version of the system. Their analysis focuses on atomicity at the level of a system-wide transaction and does not consider lower levels of atomicity. In addition, their model ignores global or persistent state whose scope is not limited to a single transaction. Similar approaches, such as the one described in [25], use stronger assumptions on the structure of the system to relax constraints on atomicity.

Other studies have used transactions or similar ideas to ensure atomicity. For example, in object-oriented communities, researchers have described approaches to update multithreaded programs and guarantee atomicity of execution [28], or proposed the use of transactions and dependency analysis for type-safe atomic updates of multiple classes [29].

Neamtiu et al. [30] introduce the notion of transactional version consistency (TVC) and describe so-termed contextual effects similar to some of those scenarios presented in our taxonomy. They recognize the need to ensure atomicity at different levels of granularity and propose a model for live update. They suggest that programmer should explicitly designate blocks of code as transactions whose execution is guaranteed to be atomic during the update process. In our analysis, we show that the level of atomicity and the constraints required at update time depend on the nature of the update itself. Hard-coding those constraints at design time is likely to be overly complicated, reduce flexibility, and hamper software evolution.

In prior work, Neamtiu et al. also describe Ginseng [16], a complete live update soltions for C programs. Ginseng supports single-threaded C programs and employs source-to-source transformation at compilation time to add indirection for functions and data types. In this case, they do not address transactional version consistentcy but restrict the solution to programmer-annotated safe update points that are still hard-coded in the original version. Static analysis is used to ensure type-safe live updates.

Hicks [31] proposes a similar approach to update programs written in Popcorn (a C-like type-safe language). Update patches are automatically generated from two versions of the source code and contain initialization and state transfer routines. Patches are then compiled into native verifiable code and dynamically linked to the running program. As before, programmers are required to annotate safe update points in the original code.

Other approaches propose static analysis to improve update safety. For example, OPUS [32] uses static analysis to warn programmers when changes to programs are likely to result in an unsafe dynamic update. In particular, warnings are reported when an update includes modifications to nonlocal program state. Unfortunately, no other system support is provided to ensure the general validity of an update and the solution described is limited to type safety.

In another direction, Buisson and Dagnat [33] explore language support to let users specify consistency constraints at update time. Their approach is tailored to updating active code at an arbitrary moment without relying on any kind of system support. As the authors admit, this approach results in very high complexity in specifying constraints at update time.

The use of user-specified update constraints is not entirely new and was first explored by Lee [34]. Lee de-

scribes DYNAMOS, one of the earliest live update solutions, comprising a toolchain and a runtime system to support live update. Updates are initiated through special user-issued commands, and users can specify what to update and what procedures must be idle at update time. Unfortunately, due to the low level of granularity, a costly live update infrastructure is necessary to synchronize access to each function call and the complexity of update constraints rapidly increases with the size of the system and the severity of an update.

To conclude our analysis, we also consider research that has proposed a particular system design to support live update. Many relevant studies can be found in the area of extensible operating systems. For example, operating systems like SPIN [35], Synthetix [36], and VINO [37] allow applications to modify kernel policies or specialize OS components. These systems are primarily concerned with enabling application-specific customizations and performance optimizations. More recently, Baumann et al. [18] describe a complete live update solution for the K42 operating system. Building on K42's object-oriented design, they use hot-swappable objects to support live update at the object level. When an object must be updated, all the incoming requests are queued until no thread is actively executing the object code. At that point, state transfer takes place and all the pending requests are redirected to the new object. Their design uses short-lived kernel threads to avoid starvation during the update process. Type safety is structurally supported but the general validity of an update is not addressed. They also assume that the author of the update must recognize and reject multi-object updates with interdependencies that may lead to a deadlock at update time.

The vast majority of the other approaches described in the literature do not address in detail consistency problems or the validity of an update in general. Most work limits the analysis to type safety and generally disallows updates to active code [38, 39, 40] or permits cross-version execution [17, 41, 42]. In both cases, it is explicitly or implicitly assumed that interleaving code from two different versions of the system does not affect the overall validity of execution. Unfortunately, no method of validation or system support is provided to verify this assumption in practice. The interested reader is referred to detailed live update surveys in [31, 43, 44, 45].

6 Conclusions

Despite being a promising solution to mitigate maintenance downtime in systems that require nonstop operation, live update is still largely perceived as an obscure niche not ready for real-life application. Many practical properties and limitations of live update are still ill-understood and have arguably not received the required attention in the literature.

In this paper, we have presented a taxonomy of live updates and proposed concrete examples to uncover those characteristics. We have discussed different scenarios with an increasing level of severity and analyzed implications for the live update process and issues in designing dependable live update infrastructures. From our analysis, an important aspect emerges: the nature of an update is central in designing systems that support live update with strong safety and predictability guarantees.

We have discussed shortcomings in existing live update solutions and proposed a new update-centric model, where the system is receptive to changes and programmers collaborate to the common intent. This vision can only be realized if the system is designed to be live updatable and each update carries with it adequate information to determine what changed and when it can be applied. In our model, feasibility, predictability, and safety of a live update are dealt with at design time, during the software development process.

We believe that software systems should be specifically conceived with live update in mind to support truly dependable live update platforms and encourage system administrators to perform maintenance activities online. We hope that much more effort will be spent on improving the safety and predictability properties of current live update solutions. The job will be finished when live update technologies can provide dependability guarantees comparable to those of conventional software updates.

7 Acknowledgments

This work has been supported by The European Research Council under grant ERC Advanced Grant 227874.

References

[1] J. Gray and D. P. Siewiorek, "High-Availability computer systems," *IEEE Computer*, vol. 24, pp. 39–48, 1991.

[2] D. A. Patterson, "A simple way to estimate the cost of downtime," in *Proc. of the 16th USENIX Systems Administration Conf.*, pp. 185–188, 2002.

[3] A. Fox, "When does fast recovery trump high reliability?," in *Proc. of the Second Workshop on Evaluating and Architecting System Dependability*, 2002.

[4] K. Poulsen, "Software bug contributed to blackout," *Security Focus*, Feb. 2004.

[5] M. Blair, S. Obenski, and P. Bridickas, "Patriot missile defense: Software problem led to system failure at Dhahran," Tech. Rep. GAO/IMTEC-92-26, United States-General Accounting Office-Information Management and Technology Division, 1992.

[6] P. Grubb and A. A. Takang, *Software maintenance: Concepts and practice*. World Scientific, 2nd ed., 2003.

[7] T. J. Ostrand and E. J. Weyuker, "The distribution of faults in a large industrial software system," *ACM SIGSOFT Softw. Eng. Notes*, vol. 27, no. 4, pp. 55–64, 2002.

[8] D. E. Lowell, Y. Saito, and E. J. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," *ACM SIGPLAN Notices*, vol. 39, no. 11, pp. 211–223, 2004.

[9] E. Rescorla, "Security holes... who cares?," in *Proc. of the 12th USENIX Security Symp.*, vol. 12, pp. 6–6, 2003.

[10] W. A. Arbaugh, W. L. Fithen, and J. McHugh, "Windows of vulnerability: A case study analysis," *IEEE Computer*, vol. 33, no. 12, pp. 52–59, 2000.

[11] C. A. N. Soules, D. D. Silva, M. Auslander, G. R. Ganger, and M. Ostrowski, "System support for online reconfiguration," in *Proc. of the USENIX Annual Tech. Conf.*, pp. 141–154, 2003.

[12] D. Pescovitz, "Monsters in a box," *Wired*, Dec. 2000.

[13] T. Dumitras, J. Tan, Z. Gho, and P. Narasimhan, "No more HotDependencies: Toward dependency-agnostic online upgrades in distributed systems," in *Proc. of the Third Workshop on Hot Topics in System Dependability*, p. 14, 2007.

[14] S. Potter and J. Nieh, "Reducing downtime due to system maintenance and upgrades," in *Proc. of the 19th USENIX Systems Administration Conf.*, pp. 6–6, 2005.

[15] A. Chou, J. Yang, B. Chelf, S. Hallem, and D. Engler, "An empirical study of operating systems errors," in *Proc. of the 18th ACM Symp. on Oper. Systems Prin.*, pp. 73–88, 2001.

[16] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," *ACM SIGPLAN Notices*, vol. 41, no. 6, pp. 72–83, 2006.

[17] K. Makris and K. D. Ryu, "Dynamic and adaptive updates of non-quiescent subsystems in commodity operating system kernels," in *Proc. of the Second ACM SIGOPS/EuroSys European Conf. on Computer Systems*, pp. 327–340, 2007.

[18] A. Baumann, J. Appavoo, R. W. Wisniewski, D. D. Silva, O. Krieger, and G. Heiser, "Reboots are for hardware: Challenges and solutions to updating an operating system on the fly," in *Proc. of the USENIX Annual Tech. Conf.*, pp. 1–14, 2007.

[19] J. P. A. Almeida, M. V. Sinderen, and L. Nieuwenhuis, "Transparent dynamic reconfiguration for CORBA," in *Proc. of the Third Int'l Symp. on Distributed Objects and Applications*, pp. 197–207, 2001.

[20] S. Ajmani, B. Liskov, and L. Shrira, "Scheduling and simulation: How to upgrade distributed systems," in *Proc. of the Ninth Workshop on Hot Topics in Operating Systems*, vol. 9, pp. 43–48, 2003.

[21] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright, "Timing the application of security patches for optimal uptime," in *Proc. of the 16th USENIX Systems Administration Conf.*, pp. 233–242, 2002.

[22] K. Makris and R. Bazzi, "Immediate multi-threaded dynamic software updates using stack reconstruction," Tech. Rep. TR-08-007, Arizona State University, 2008.

[23] D. Gupta, P. Jalote, and G. Barua, "A formal framework for on-line software version change," *IEEE Trans. Softw. Eng.*, vol. 22, no. 2, pp. 120–131, 1996.

[24] J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Trans. Softw. Eng.*, vol. 16, no. 11, pp. 1293–1306, 1990.

[25] Y. Vandewoude, P. Ebraert, Y. Berbers, and T. D'Hondt, "Tranquility: A low disruptive alternative to quiescence for ensuring safe dynamic updates," *IEEE Trans. Softw. Eng.*, vol. 33, no. 12, pp. 856–868, 2007.

[26] M. Niamanesh, N. F. Nobakht, R. Jalili, and F. H. Dehkordi, "On validity assurance of dynamic reconfiguration for component-based programs," *Electronic Notes in Theoretical Computer Science*, vol. 159, pp. 227–239, 2006.

[27] T. Bloom and M. Day, "Reconfiguration and module replacement in Argus: Theory and practice," *Software Engineering J.*, vol. 8, no. 2, pp. 102–108, 1993.

[28] Y. Murarka and U. Bellur, "Correctness of request executions in online updates of concurrent object oriented programs," in *Proc. of the 15th Asia-Pacific Software Engineering Conf.*, pp. 93–100, 2008.

[29] S. Zhang and L. Huang, "Type-Safe dynamic update transaction," in *Proc. of the 31st Annual Int'l Computer Software and Applications Conf.*, vol. 2, pp. 335–340, 2007.

[30] I. Neamtiu, M. Hicks, J. S. Foster, and P. Pratikakis, "Contextual effects for version-consistent dynamic software updating and safe concurrent programming," in *Proc. of the 35th Annual ACM SIGPLAN-SIGACT Symp. on Principles of Programming Languages*, pp. 37–49, 2008.

[31] M. Hicks, *Dynamic software updating*. PhD thesis, University of Pennsylvania, 2001.

[32] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz, "OPUS: Online patches and updates for security," in *Proc. of the 14th USENIX Security Symp.*, vol. 14, pp. 19–19, 2005.

[33] J. Buisson and F. Dagnat, "Introspecting continuations in order to update active code," in *Proc. of the First Int'l Workshop on Hot Topics in Software Upgrades*, pp. 1–5, 2008.

[34] I. Lee, *Dymos: A dynamic modification system*. PhD thesis, The University of Wisconsin-Madison, 1983.

[35] B. N. Bershad, S. Savage, E. G. Sirer, M. E. Fiuczynski, D. Becker, C. Chambers, and S. Eggers, "Extensibility, safety and performance in the SPIN operating system," in *Proc. of the 15th ACM Symp. on Oper. Systems Prin.*, vol. 29, pp. 267–284, 1995.

[36] C. Cowan, T. Autrey, C. Krasic, C. Pu, and J. Walpole, "Fast concurrent dynamic linking for an adaptive operating system," in *Proc. of the Third Int'l Conf. on Configurable Distributed Systems*, pp. 108–115, 1996.

[37] M. Seltzer and C. Small, "Self-Monitoring and self-adapting operating systems," in *Proc. of the Sixth Workshop on Hot Topics in Operating Systems*, pp. 124–129, 1997.

[38] O. Frieder and M. E. Segal, "On dynamically updating a computer program: From concept to prototype," *J. Syst. Softw.*, vol. 14, no. 2, pp. 111–128, 1991.

[39] D. Gupta and P. Jalote, "On line software version change using state transfer between processes," *Softw. Pract. and Exper.*, vol. 23, no. 9, pp. 949–964, 1993.

[40] J. Arnold and M. F. Kaashoek, "Ksplice: Automatic rebootless kernel updates," in *Proc. of the Fourth ACM European Conf. on Computer systems*, pp. 187–198, 2009.

[41] H. Chen, J. Yu, R. Chen, B. Zang, and P. Yew, "POLUS: A POwerful live updating system," in *Proc. of the 29th Int'l Conf. on Software Engineering*, pp. 271–281, 2007.

[42] H. Chen, R. Chen, F. Zhang, B. Zang, and P. Yew, "Live updating operating systems using virtualization," in *Proc. of the Second Int'l Conf. on Virtual Execution Environments*, pp. 35–44, 2006.

[43] S. Ajmani, "A review of software upgrade techniques for distributed systems," 2004.

[44] D. Gupta, *On-Line software version change*. PhD thesis, Indian Institute of Technology, 1994.

[45] M. E. Segal and O. Frieder, "On-the-Fly program modification: Systems for dynamic updating," *IEEE Softw.*, vol. 10, no. 2, pp. 53–65, 1993.