A Taxonomy of Live Updates

Cristiano Giuffrida, Andrew S. Tanenbaum

Department of Computer Science
Vrije Universiteit
Amsterdam, The Netherlands
{giuffrida,ast}@cs.vu.nl

Keywords: Live Update, Dynamic Software Updating, Dependability.

Abstract

Many high-availability systems require regular software updates but can hardly afford any downtime. Existing general-purpose live update approaches proposed as a solution to this problem have failed to reach broad acceptance.
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. To substantiate our claim, 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.

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 5 minutes of downtime per year [11]. In other applications, such as factories and power plants, availability constraints are even more tight.
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 [12]. End users have learned to live with the many changes that programs undergo over time. The introduction of new features, enhancements, bug fixes and security patches are the norm rather then the exception in widely adopted software solutions.
Traditionally, an update requires a full system reboot before the changes can take effect, which constitutes a major problem for systems that must provide strong availability guarantees.

1.1 Live Update

Live update—namely the ability to update software without service interruption—is a promising direction to support software evolution in high-availability environments. 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 updatable software systems have also other potential benefits [25]. First off, live update offers a higher degree of flexibility. 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 or to dynamically address the behavior of a running system.

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 general-purpose live update technologies have failed to receive broad acceptance.

Second, we propose a taxonomy of live updates focusing on the nature of the update. 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.
Finally, we discuss a model for update-aware systems that can support dependable live update by design.

2 Models for Live Update

Live update has received a great deal of attention in the past two decades. We concentrate here on software-based solutions, which aim to provide runtime support to apply online changes to running software. These approaches can be 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 [8] or using virtualization technologies [17, 24] have also been explored. These approaches, however, require ad-hoc infrastructures—a separate execution environment or virtual machines—and are usually more tailored to full system updates.
In the present paper, we mainly focus on live update for operating systems (OSes). There are a number of reasons to concentrate our attention on operating systems. First, OSes are plagued with continuous maintenance updates. Second, high availability of operating systems is a major concern. Downtime in an OS directly translates to downtime for all the hosted applications. Finally, operating systems offer a complete set of functionalities and update scenarios. This is useful for our analysis. Furthermore, system administrators, programmers, and other categories of users are already familiar with many real-life examples of OS updates.
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 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—e.g. linking a component in a component-based system. 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 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, such as: function pointers [22], dynamic instrumentation [19], indirection tables [4], interceptors and naming services [1]. The live update framework incorporates the ability to compare the old version and the new version of the system to figure out how to apply changes correctly. Using these techniques, execution is resumed on the new version seamlessly, without the original version being aware of the update.

As a result of this model, much attention has been dedicated to backward compatibility and transparency. These properties have been largely promoted as key success criteria for live update technologies. Unfortunately, despite the ability to support legacy systems, assuming that the live update infrastructure is invisible to the development process delegates to the infrastructure the entire responsibility of applying the update and ensuring that the resulting configuration is valid. This process is generally complicated and error-prone.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.

Due to the high complexity involved, most live update solutions do not attempt to address the general safety of an update [18]. In the literature, the role of safety constraints in live update solutions is controversial. The reason for this lies at the theoretical foundations of live update. Pioneering work on the validity of a live update was undertaken by Gupta [14] 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.

Other approaches specific to event-driven systems have suggested using atomicity at the level of an event to make sure that each event-generated transaction is entirely executed on a single version of the system [16]. Supporting only atomicity at the level of an event, however, reduces the degree of flexibility,

since atomicity constraints cannot be adapted to each particular update. This may cause excessive system disruption for small updates [26], as well as hamper the ability to support more complex updates that require stronger atomicity guarantees.

3 The Taxonomy

In this section, we propose a taxonomy of live updates and show that, in each scenario, the nature of the update is crucial to determine the properties of the live update process and the impact on a running system. Rather than focusing on formal definitions or striving for completeness, we propose criteria to describe the nature of an update and discuss possible scenarios through examples. 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 OS 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 the structural units, 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 OS 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 the following 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 bound checking on a string to avoid buffer overflow attacks. An example of performance improvement 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.

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 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 is 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 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 OSes. 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 origi-

nal data block may have been lost. In contrast, if we allowed the update at a time when neither protocols 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. 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 OS 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. 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. 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 system configuration. In many 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.

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 easily established.

In addition, our investigation shows that the dominant assumptions used in the literature may lead to undesirable effects. In particular, restricting the design to support only atomicity at the event level will result in reduced flexibility with important consequences. For example, using *quiescence* [16] as the only stability condition is unnecessarily expensive in the average case, as also argued in [26]. In highly connected systems such as OSes, 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 results in poor stability guarantees. 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 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.

4.1 Towards Update-aware Systems

In the previous sections, we argued that the nature of the update is crucial to build a dependable live update solution. We believe the system should support live update by design and published updates should contain more information about what they affect and how, thus allowing the system to apply them safely at the right time. Note that we are talking about both small security patches as well as functional changes from one version to the next.

To address this challenge, in our prior work, we have presented a new model for dependable live update, in which the system is update-aware and cooperates during the update process [10]. The system is designed to be highly modular and broken down into separate components. The software developers producing the update, in turn, must include both code changes and update constraints in a *live update package* of a predetermined format. The update constraints are directives the system must satisfy at update time to apply changes online in a reliable way.

When an update becomes available, the system allows an update manager to notify all the components that must be replaced and ask them to converge to a particular state as required by the update. Each component supports by design a number of states it is able to converge to in bounded time. The update constraints included in the live update package must specify the state each affected component has to reach before updating. When a component is ready, it will save its state in a safe place and send back to the update manager a *ready* 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.

It is easy to show that this update-centric model results in higher flexibility and solves the problem of establishing a safe update time structurally. For example, recall the scenario described in the taxonomy when a protocol between two components (e.g. the virtual filesystem and the disk driver) changes. The software developers must include the new version of the two components in the package and, at the same time, specify a safe state for each component at update time. For example, an appropriate state for both components may be *no disk I/O in progress*. As a result, when the two components are replaced at the end of the update process, no protocol will be in progress and state transfer will probably not even be necessary. In the opposite direction, an update including a small security fix for a particular component can probably be applied almost immediately with minimal service disruption if the state required at update time is simply *no activity in progress*.

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 [19].

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 [14] and other researchers [23] 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 [5] 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 [16] 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 [26], 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 [20], or proposed the use of transactions and dependency analysis for type-safe atomic updates of multiple classes [27].

Neamtiu et al. [21] 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 complicated and also hamper software evolution.

In prior work, Neamtiu et al. also describe Ginseng [22], a complete live update solution for C programs. In this case, they do not address transactional version consistency 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 [15] 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 [2] 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.

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 [3, 9, 13] or permits cross-version execution [6, 7, 19]. 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.

6 Conclusions

Despite being a promising solution to mitigate maintenance downtime in systems that require nonstop operation, general-purpose live update is still largely perceived as an obscure niche by most end users. 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 justified the need for 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. For the update-aware systems we envision, feasibility, predictability, and safety of a live update are dealt with at design time, during the software development process.

7 Acknowledgments

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

References

[1] 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*, pages 197–207, 2001.

[2] G. Altekar, I. Bagrak, P. Burstein, and A. Schultz. OPUS: Online patches and updates for security. In *Proc. of the 14th USENIX Security Symp.*, volume 14, pages 19–19, 2005.

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

[4] 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.*, pages 1–14, 2007.

[5] T. Bloom and M. Day. Reconfiguration and module replacement in Argus: Theory and practice. *Software Engineering J.*, 8(2):102–108, 1993.

[6] 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*, pages 35–44, 2006.

[7] 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*, pages 271–281, 2007.

[8] 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*, page 14, 2007.

[9] O. Frieder and M. E. Segal. On dynamically updating a computer program: From concept to prototype. *J. Syst. Softw.*, 14(2):111–128, 1991.

[10] C. Giuffrida and A. S. Tanenbaum. Cooperative update: a new model for dependable live update. In *Proceedings of the 2nd International Workshop on Hot Topics in Software Upgrades*, pages 1–6, 2009.

[11] J. Gray and D. P. Siewiorek. High-Availability computer systems. *IEEE Computer*, 24:39–48, 1991.

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

[13] D. Gupta and P. Jalote. On line software version change using state transfer between processes. *Softw. Pract. and Exper.*, 23(9):949–964, 1993.

[14] D. Gupta, P. Jalote, and G. Barua. A formal framework for on-line software version change. *IEEE Trans. Softw. Eng.*, 22(2):120–131, 1996.

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

[16] J. Kramer and J. Magee. The evolving philosophers problem: Dynamic change management. *IEEE Trans. Softw. Eng.*, 16(11):1293–1306, 1990.

[17] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable virtual machines enabling general, single-node, online maintenance. *ACM SIGPLAN Notices*, 39(11):211–223, 2004.

[18] K. Makris and R. Bazzi. Immediate multi-threaded dynamic software updates using stack reconstruction. Technical Report TR-08-007, Arizona State University, 2008.

[19] 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*, pages 327–340, 2007.

[20] 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.*, pages 93–100, 2008.

[21] 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*, pages 37–49, 2008.

[22] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. *ACM SIGPLAN Notices*, 41(6):72–83, 2006.

[23] 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*, 159:227–239, 2006.

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

[25] 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.*, pages 141–154, 2003.

[26] 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.*, 33(12):856–868, 2007.

[27] S. Zhang and L. Huang. Type-Safe dynamic update transaction. In *Proc. of the 31st Annual Int'l Computer Software and Applications Conf.*, volume 2, pages 335–340, 2007.