To Upgrade or Not to Upgrade

Impact of Online Upgrades across Multiple Administrative Domains

Tudor Dumitraş
Carnegie Mellon University
tudor@cmu.edu

Priya Narasimhan
Carnegie Mellon University
priya@cs.cmu.edu

Eli Tilevich
Virginia Tech
tilevich@cs.vt.edu

Abstract

Online software upgrades are often plagued by runtime behaviors that are poorly understood and difficult to ascertain. For example, the interactions among multiple versions of the software expose the system to race conditions that can introduce latent errors or data corruption. Moreover, industry trends suggest that online upgrades are currently needed in large-scale enterprise systems, which often span *multiple administrative domains* (e.g., Web 2.0 applications that rely on AJAX client-side code or systems that lease cloud-computing resources). In such systems, the enterprise does not control all the tiers of the system and cannot coordinate the upgrade process, making existing techniques inadequate to prevent *mixed-version races*. In this paper, we present an analytical framework for impact assessment, which allows system administrators to directly compare the risk of following an online-upgrade plan with the risk of delaying or canceling the upgrade. We also describe an executable model that implements our formal impact assessment and enables a systematic approach for deciding whether an online upgrade is appropriate. Our model provides a method of last resort for avoiding undesirable program behaviors, in situations where mixed-version races cannot be avoided through other technical means.

Categories and Subject Descriptors D.2.7 [*Software Engineering*]: Distribution, Maintenance, and Enhancement; K.6.3 [*Management of Computing and Information Systems*]: Software Management; C.2.4 [*Computer-Communication Networks*]: Distributed Systems

General Terms Management, Reliability

Keywords Mixed-version race, Online upgrade, Multiple administrative domains, Risk assessment

1. Introduction

Actively used software must be modified continuously to ensure its utility and safety. Fixing bugs, adding new features, removing obsolete features, optimizing performance—all involve upgrading existing software systems. Moreover, current industry trends suggest that upgrade-related downtime is unacceptable for many large-scale distributed systems, such as electrical utilities, assembly-line manufacturing, customer support, e-commerce or online banking [6]. These systems must employ online-upgrade techniques.

During an online upgrade, the system enters states that emerge at runtime and that may not have been validated in advance. Multi-tier enterprise systems often employ *rolling upgrades*, which upgrade-and-reboot each node at a time, in a wave rolling through the distributed system. Rolling upgrades place the system in a state with *mixed versions*, where requests might be processed by either the old or the new version during the upgrade. In general, the behavior of a system with mixed versions is not guaranteed to conform to the specification of either version of the software and is hard to validate in advance [25].

For example, while the new version can be backward-compatible, the old version can not handle invocations that require the new version's semantics. Previous research advocates coordinating the upgrade operations [14, 26, 29], to prevent the new version from calling into the old version, simulating the interfaces of past and future versions during the upgrade [1] or performing upgrades atomically, end-to-end [10], to avoid mixed versions altogether. These approaches are infeasible in large-scale distributed systems that span *multiple administrative domains* (e.g., by relying on client-side code or on cloud-computing resources), where an online upgrade's administrator does not control all the tiers and cannot coordinate their upgrades. We show that, in systems that communicate across administrative domains using asynchronous messaging, a rolling upgrade exposes such a system to a new kind of race condition, involving multiple software versions.

Such *mixed-version races* might be benign, but they might also have a critical impact (in 1994, a similar condition in a banking system caused a $15M loss for the bank's customers in a single day [12]). Conversely, delaying the upgrade of a system with known software defects might also have a negative impact. The trade-offs between upgrading and not upgrading are not easy to ascertain.

We are aware of two real-world upgrade failures that can be traced to mixed-version races [12, 23]. However, to the best of our knowledge, this race condition has not been described before in the research literature.

Instead of preventing mixed-version races, or other unexpected behaviors that can result from an online upgrade, we propose assessing the risk they pose to the system. Our model helps answer the following question: *Is it worth suffering a potential inconsistency during an online upgrade in order to introduce a change that addresses a known issue in the running system?* Addressing an issue encompasses corrective and perfective maintenance [28], i.e., fixing software defects and adding new features, respectively. While bugs and upgrade inconsistencies are both undesirable, answering this question allows developers and administrators to choose the lesser evil.

We have devised a comprehensive model taking into consideration all the parameters that influence the risks of bugs and mixed-version races. These parameters include the time needed to upgrade a single host, the number of hosts to upgrade in a certain tier of the system, and the number of messages exchanged between tiers.

We believe that, for a risk-assessment method to be useful, it must not require testing the entire mixed-version state space, which exhibits combinatorial explosion. Therefore, by understanding the sequence of events that exposes the race conditions, we assess their impact in a limited number of system configurations, and we derive the overall risk of upgrading analytically.

Through three case studies of upgrades—performed in both mission-critical systems (online banking) and in Internet services with relaxed consistency requirements (social networking)—we emphasize that our model makes it possible to concretely quantify these risks. In fact, our model commonly recommends counter-intuitive, but correct, decisions. We have also created an executable model, in the form of a Web application, to reify our risk-assessment approach and to demonstrate that it can benefit real users. Risk assessment represents a method of last resort, for the situations where mixed-version races cannot be avoided through other technical means.

This paper makes three contributions:

- We describe mixed-version races, which can occur during upgrades across multiple administrative domains, and we identify the system interactions that lead to such race conditions;
- We develop an analytical framework for reasoning about the trade-off between upgrading in the presence of mixed-version races and delaying an upgrade that corrects known software defects;
- We present an online tool that implements our analytical model and we demonstrate its use, on three case studies, to decide whether *to upgrade or not to upgrade*.

We note that, in practice, the risk of upgrading can be influenced by additional factors, such as known bugs in the new version. In this paper, we focus on assessing the impact of mixed-version races. A comparison between the risks introduced by bugs in the old and new versions can be achieved through known testing methods and is outside the scope of this paper.

The rest of this paper is structured as follows. In Section 2, we review the state of the art in benchmarking the dependability of systems that undergo online upgrades. In Section 3, we introduce mixed-version races. In Section 4, we formally present our analytical risk model. In Section 5, we describe three case studies of online upgrades, and in Section 6 we discuss the implications of our contribution.

2. Background

Historically developed in the telecommunications industry for the maintenance of telephone switches, *dynamic software updating* techniques [13] focus on upgrading single-node systems on the fly. However, industry trends suggest that online-upgrade techniques are currently needed in a wide range of distributed systems (e.g., electrical utilities, assembly-line manufacturing, customer support, e-commerce, online banking) [6]. The characteristics of distributed systems simplify some aspects of the upgrade problem, while complicating others. Specifically, while distributed systems include redundancy and fault-tolerance mechanisms, allowing components to be temporarily inaccessible, they also require more complex interactions among the heterogeneous system components (e.g., asynchronous messaging, long-running transactions, reads/writes to shared storage). Moreover, in distributed systems spanning multiple administrative domains, it may be difficult to coordinate the operations performed during an online upgrade.

Online upgrades in distributed systems. The earliest work on distributed-system upgrades relies on the crash recovery and state transfer mechanisms from the Argus system [3], which were originally developed for coping with crash faults and network partitions. Similarly, the Eternal system, which provides fault tolerance to legacy CORBA applications by redirecting the message exchanges to a group-communication protocol, leverages this mechanism to coordinate the distributed upgrade [29]. The authors observe, however, that certain communication patterns used in practice, such as one-way or asynchronous messages, prevent Eternal from enforcing the quiescence needed for upgrading the CORBA objects that receive these messages.

The Conic system [14] upgrades component-based systems through architectural reconfigurations (i.e., changing components and connectors) and can achieve quiescence if each component provides a minimal control API: passivate, assert(active/passive), activate, link, unlink. Conic determines the correct sequence of control API calls required when upgrading a component (e.g., passivating all its inbound nodes). These principles are reflected in modern component frameworks such as R-OSGi, which upgrade a component along with the transitive closure of its inbound dependencies [24].

In the absence of fault-tolerance mechanisms or control APIs, the PODUS system establishes simple rules for coordinating a distributed-system upgrade, such as upgrading servers before their clients [26]. This approach can be extended to systems that communicate across multiple administrative domains using remote procedure calls (RPC), which consist of synchronous request-and-reply message exchanges. Instead of strictly enforcing the order of local upgrades, the Upstart system [1] enables a mixed-version operating mode by providing simulation objects, which implement the interfaces of past and future versions. This approach requires disallowing some incompatible invocations during the distributed-system upgrade.

We previously developed the Imago system [10], which upgrades distributed systems atomically end-to-end, and we showed that this approach improves the dependability of online upgrades. In particular, atomicity prevents the mixed-version races introduced in this paper. In large-scale distributed systems, which often span multiple administrative domains, we must reason about the impact of relaxing the atomicity guarantees on system dependability.

Industry best-practices for online upgrade. Prior research suggests that there is a tension between between the upgrade atomicity and the system availability during the upgrade. System administrators sometimes favor the atomicity, by upgrading inter-dependent components together, during windows of planned downtime [9], or by placing the old version in a read-only mode during the upgrade. However, many enterprises can no longer afford the high cost of downtime and must upgrade their systems online, without constraining the live workload [6, 22]. Industry best-practices recommend *rolling upgrades*, which upgrade-and-reboot each node in a wave rolling through the cluster [5, 18]. A rolling upgrade avoids downtime and imposes very little capacity loss, but it requires the old and new versions to interact with each other in a compatible manner. Moreover, new features introduced by an upgrade sometimes require the system operators to undergo a lengthy re-training process, which mandates a gradual deployment of the new version at different sites [9]. In such cases, the enterprise application will include a mix of versions that operate concurrently at different installation sites, without placing any site in a read-only mode and without allowing state divergence.

These requirements have motivated the introduction of several commercial products for synchronizing the persistent state of two versions [6] and for performing rolling upgrades [16, 21]. However, these commercial products provide no way of determining if the interactions between mixed versions are safe and leave these concerns to the application developers. Moreover, rolling upgrades can lead to mixed-version races [23].

Dependability of online-upgrade techniques. Evaluations of the previous upgrade mechanisms typically focus on the range of updates (i.e., the types of changes supported) and on the overhead imposed, rather than on the upgrade dependability. Field studies [2, 20], surveys [7, 19], fault injection [10, 19] and direct experimentation [7, 31], have been used to assess the effectiveness of previous approaches in reducing the number of upgrade failures.

Beattie et al. [2] analyze the security patches released between 1999–2001 and recorded in a vendor-independent database, and they find that software defects were discovered in 18% of these patches. Oppenheimer et al. [20] study the failures recorded by three large-scale Internet services, and they report that 4.6–10 component failures and 0.7–6 system-wide failures occur each month, mostly during regular maintenance activities. Oliveira et al. [19] present a survey of 51 database administrators, who report eight classes of faults: deployment, performance, general-structure, DBMS, access-privilege, space, general-maintenance, and hardware. Crameri et al. [7] present a similar survey, of 50 system administrators, who report that the average and maximum failure rates for upgrades, in their infrastructures, are 8.6% and 50%, respectively. We previously developed Ecotopia [11], a framework for scheduling change-management operations in complex service-oriented architectures (SOA) by asking "what-if" questions about the impact of operations that span multiple administrative domains. Zheng et al. [31] propose running experiments with different configurations, in a virtualized data center, in order to reduce the cost of answering "what-if" questions. We introduced an upgrade-centric fault model [10], with four fault types, and proposed benchmarking the dependability of online-upgrade techniques through fault-injection experiments driven by our fault model.

To the best of our knowledge, ours is the first description of *mixed-version races*, a new kind of race condition that a rolling upgrade can expose in a large-scale distributed system spanning multiple administrative domains. We therefore take a different approach than the prior work. Instead of preventing mixed-version races through a new technique—which might be infeasible under the realistic assumptions of the systems that we target—we present the best possible alternative, risk assessment. Unlike the previous approaches for evaluating the dependability of online upgrades, our risk model does not rely on field or experimental data. We make use of system parameters and testing results that are readily available to the developers and administrators. By quantifying the trade-off between upgrading in the presence of mixed-version races and delaying an upgrade that corrects known software defects, this work can help upgrade administrators make informed decisions regarding whether *to upgrade or not to upgrade*.

3. Mixed-version races

Rolling upgrades, which gradually upgrade each node in the cluster, are widely believed to reduce the risks of upgrading because failures are localized and might not affect the entire distributed system [9, 21]. However, they also create states with mixed versions, which expose the system to a new type of race condition, which we call *mixed-version race*.

Mixed-version races occur in systems that span multiple administrative domains, where a consistent upgrade schedule cannot be enforced. Asynchronous message exchanges across domain boundaries potentially lead to a situation where a callback from the new version is processed by the old version on a different tier of the application.

We illustrate mixed-version races with an online banking example. Banks are starting to employ online upgrades [6], in spite of the inherent risks of data inconsistency associated with current upgrading approaches. We consider an online banking application that uses the AJAX style of web programming, where part of the application code is executed at client-side, in multiple web browsers.

The following sequence of events leads to a mixed-version race (see also Figure 1):

1. The bank initiates a rolling upgrade of its infrastructure. The rolling upgrade places the system in a state where two versions (old and new) co-exist in the front-end. Both versions handle client requests, during the upgrade.
2. The bank customer starts an online banking session. Her browser sends an initial request to load the front page of the banking application.
3. The request arrives at a front-end server that was already upgraded and that runs the new version. The user's browser loads the new version of the web page, which includes both static HTML markup and Javascript code. This code implements the client side functionality of the application.
4. The user initiates an operation that requires additional communication with the server. Rather than reloading an entire page, the client-side code issues an `XMLHttpRequest` callback into the server, to reload part of the banking page that is currently displayed.
5. The asynchronous callback, which was issued by the new version of the client-side code, arrives at a server that was not yet upgraded.The old version of the server-side code does not know how to handle the request and throws an exception (in the best case) or handles the request incorrectly (in the worst case).
6. When the user receives the reply, she may or may not notice that an error has occurred.

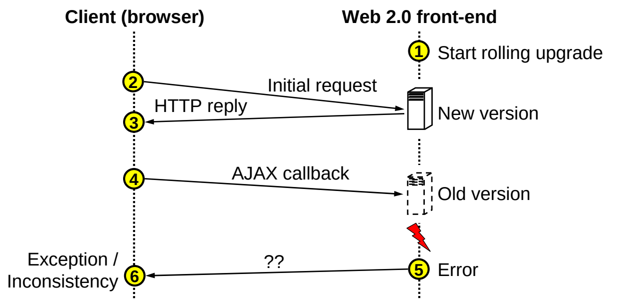


Figure 1. Mixed-version race.

If the web front-end includes only a few servers, which can be upgraded quickly, the window of vulnerability to mixed-version races is small. However, these race conditions occur frequently during rolling upgrades of large Internet systems, such as Facebook [23].

For banking applications, the inconsistencies that may result can have severe consequences, including financial losses. For example, if the code that checks whether to allow a cash transfer is moved from the server-side to the client-side (e.g., in order to push some computational load to the clients), a mixed-version race can lead to this code executing twice. In this situation, a request to debit \$100 from a bank account would subtract \$200 from the user's account balance because of the double invocation of the debit operation: once from the browser and once from the server.

In 1994, a similar upgrade of Chemical Bank's data center affected more than 100,000 customers over the course of a single day. Each ATM withdrawal was deducted twice from the customer's account, adding up to a \$15M loss. Moreover, some checks bounced, which made Chemical Bank customers incur additional fees at other financial institutions. The upgrade changed a single line of code in the server-side software [12].

3.1 Key technical challenges

The mixed-version race described above could have been avoided by extending the load balancer, which dispatches client requests to the front-end servers, to track the progress of the rolling upgrade and to determine the appropriate server-side version for each request. This approach would require adding significant complexity and processing delays to a key component of the enterprise infrastructure, which is essential for avoiding performance bottlenecks. Alternatively, the servers could wait until the end of the rolling upgrade before starting to send the new version of the client-side Javascript code. In a large enterprise infrastructure, where some servers are likely to become unresponsive during the upgrade—either because they have failed or because they are slow to upgrade—it is difficult to determine reliably when the rolling upgrade has completed. Prior anecdotal evidence, from the recorded occurrences of mixed-version races [12, 23], confirms that these conditions cannot be avoided easily.

There are three technical challenges that render mixed-version races hard to address using existing techniques:

- **Non-atomic upgrades.** A rolling upgrade is not an atomic operation, and it places the system in a state with mixed versions. In large-scale infrastructures, some nodes crash during the upgrade and other nodes need a long time to complete the upgrade. Moreover, some upgrades fail silently. In such an environment, the end of the rolling upgrade is not always easy to detect because it is hard to distinguish a node that has crashed from a node that is slow. Because the upgrade is a long-running procedure, often enterprises cannot delay exposing the new functionality to the other tiers of the application.
- **Asynchronous messaging.** Asynchronous communication is used, for performance reasons, in all the tiers of modern enterprise systems. For instance, in the front-end AJAX applications receive asynchronous callbacks from the client-side code, in the middle tier application servers use message-oriented middleware (e.g., Amazon's Simple Queue Service, XMPP), and in the back-end storage systems use asynchronous I/O. Asynchronous communication is considered by some experts a better paradigm for building distributed systems than synchronous RPC [30].
- **Versions determined dynamically.** When asynchronous message exchanges occur concurrently with long-running rolling upgrades, the code versions involved in the exchange are determined dynamically (e.g., at the time of the first invocation). Upgrades performed in the middle of the message exchange expose the system to mixed-version races.

As online-upgrade techniques are increasingly adopted by contemporary enterprise application, similar problems will become widespread. Distributed enterprise systems have been using heterogeneous, off-the-shelf components for a long time. With the advent of cloud computing, these third-party components are also provisioned and managed by third parties, such as public cloud infrastructures (e.g., the Amazon Web Services). These enterprise systems span multiple administrative domains and no longer control the upgrading schedule for all their tiers. Cloud-based resources (e.g., storage objects, message queues) are upgraded on schedules set by the service providers, and upgrades may occur during an asynchronous message exchange between tiers. In other words, third-party provisioning, despite all its benefits, will likely introduce the risk of mixed-version races for a wide range of applications.

4. Upgrade-risk model

Our model answers the question "*is it riskier to upgrade or not to upgrade?*" By combining the likelihood of mixed-version races with the severity of the resulting errors and inconsistencies—which characterizes the impact of potential upgrade failures—we estimate the *risk of upgrading*. We then compare this result with the *risk of not upgrading*, obtained from the severity of the original bugs or feature requests that are addressed by the upgrade. In other words, we estimate the expected impacts of the two alternative decisions—to upgrade or not to upgrade—over the typical time frame of a rolling upgrade.

4.1 Assumptions

Our approach is predicated by four assumptions:

- We assume that the software developers and system administrators use a uniform labeling system, which covers the severity of known bugs, the criticality of feature addition/change/removal requests, as well as the severity of the inconsistencies that might occur during an upgrade.
- We assume that a thorough integration-testing procedure is in place, and that it can be extended to the system states with mixed versions.
- We assume that the atomic unit of upgrade is the host, i.e. that all the collocated components that are upgraded concurrently are exposed to the users after the host reboots.
- We assume that, in most cases, developers and administrators cannot estimate accurately the likelihood of exposing known bugs or of invoking new callbacks or the variability of upgrade durations for each host.

Mixed-version testing is done using only two hosts, one running the new version and the other running the old version, and triggers the worst case scenario leading to a mixed-version race: a callback from the new version arriving at the old version, as described in Section 3. The inconsistencies discovered in this manner are assigned their own severity levels, and the uniform labeling system ensures that they are comparable with the impact of known bugs.

The complexity and duration of this testing procedure depends on the differences between the old and new versions, but not on the number of potential mixed-version states created at runtime. For example, out of the 352 servers supporting Wikipedia, one of the ten most popular sites on the Internet, 120 hosts are located on the front end and can be accessed by the users.[1] This could lead to $2^{120} \approx$ 1E36 (one undecillion) possible version combinations during a rolling upgrade similar to the one described in Section 3. Instead, we test only one combination.

We can extend this testing approach to upgrade scenarios where n mixed versions must coexist (with $n > 2$) or where m tiers of the distributed system are affected by the upgrade. In the first case, we have to consider all the cases where a version can invoke an older version, and we must test $\binom{n}{2} = \frac{1}{2}n(n-1)$ mixed-version combinations. In the second

[1] The information on Wikipedia dates from April 2009.

case, we must consider all the cases where the new version invokes the old version in the next tier, and we must test $(m-1)2^{m-2}$ combinations.

In practice, however, integration testing is not likely to be affected by combinatorial explosion because it is uncommon to support a large number of mixed versions and because distributed systems have only a few tiers that span multiple administrative domains (e.g., for $m = 4$ we have to test only 12 combinations). Moreover, because during a rolling upgrade each individual host is upgraded in an atomic fashion—by disconnecting, upgrading, rebooting and reintegrating the host into the distributed system—the number of collocated components that must be upgraded does not affect the complexity of the testing procedure. In this paper, we focus on the most common situation, where the system spans two administrative domains and includes only two versions during the rolling upgrade: the old version and the new version.

To enhance the usability of our analytical model, we do not use continuous probability values for expressing the likelihood of exposing bugs or mixed-version inconsistencies, because these values are difficult to estimate accurately. Instead, we use a discrete probability measure, with three possible values: low, medium, and high. Similarly, we require system administrators to specify the duration of single-host upgrades in the form of a triangular distribution, with an expected value and lower/upper bounds. In consequence, the outputs from our model are discrete values as well, which simplifies the comparison between the impacts of upgrading and of not upgrading. Working with discrete values allows administrators to capture the partial information available about the system and to use it for deciding when and how to execute an upgrade.

4.2 Analytical risk model

Table 1 describes the input and output parameters of our risk model. N_{call}, N_{bug}, c, $\mathbb{S}(\mathbb{I}_k)$ and $\mathbb{S}(\mathbb{B}_k)$, are determined through integration testing. $P_{call}(k)$ and $P_{bug}(k)$ are workload-dependent metrics, which are estimated from testing results and from system monitoring logs. U, $\overline{\tau}$, τ_{lo} and τ_{hi} are provided by the system administrators. We assess:

$$\mathbf{Risk}_{\text{no upgrade}} = \frac{\sum_{k=1}^{N_{bug}} \Pr[\mathbb{B}_k] \cdot \mathbb{S}(\mathbb{B}_k)}{N_{bug} \cdot \max \mathbb{S}}$$

$$\mathbf{Risk}_{\text{upgrade}} = \frac{\sum_{k=1}^{N_{call}} \Pr[\mathbb{I}_k] \cdot \mathbb{S}(\mathbb{I}_k)}{N_{call} \cdot \max \mathbb{S}},$$

which combine the likelihoods of inconsistencies and bug manifestations with the corresponding severity levels. We normalize the risk values with respect to $\max \mathbb{S}$ in order to keep them comparable across different severity scales.

The inputs $P_{call}(k)$ and $P_{bug}(k)$ can take one of the values p_{lo}, p_{med} or p_{hi}, which correspond to low, medium and high probabilities. These discrete levels are easier to specify than precise probability values. In our analysis, we do not attempt to assign placeholder values to these probability levels, and instead we derive the risk symbolically. To avoid counter-intuitive artifacts in the computation, we consider that p_{lo}, p_{med} or p_{hi} correspond to a linear scale, i.e., $p_{med} = 2p_{lo}$ and $p_{hi} = 3p_{lo}$.

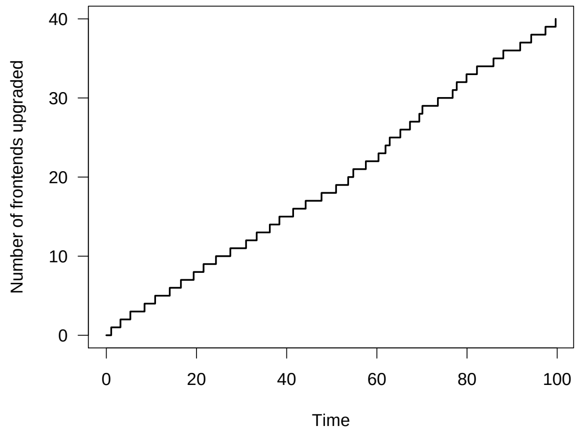
Model inputs	
U	Number of servers upgraded.
$\overline{\tau}$	Mean upgrade duration for a single host.
τ_{lo}, τ_{hi}	Lower and upper bounds for the upgrade duration.
c	Average number of callbacks per request issued by the new version of the client-side code.
N_{call}	Number of callbacks that can trigger a mixed-version race, because they do not exist in the old version or because they have different semantics.
N_{bug}	Number of bugs addressed by the upgrade.
$\mathbb{S}(\mathbb{E})$	Severity of event $\mathbb{E}$ (e.g., manifestation of bugs $\mathbb{B}_1, \mathbb{B}_2 \ldots \mathbb{B}_{N_{bug}}$ or of mixed-version inconsistencies $\mathbb{I}_1, \mathbb{I}_2 \ldots \mathbb{I}_{N_{call}}$).
$P_{call}(k)$	Probability of issuing the callback that leads to mixed-version inconsistency $\mathbb{I}_k$.
$P_{bug}(k)$	Probability that a request will expose bug $\mathbb{B}_k$.
Model outputs	
$\mathbf{Risk}_{\mathbb{D}}$	The risk associated with decision $\mathbb{D} \in \{\text{upgrade}, \text{no upgrade}\}$.
Because the risk of inconsistency varies during the upgrade, we estimate the average risk, $\overline{\mathbf{Risk}_{\text{upgrade}}}$, and the maximum risk, $\max(\mathbf{Risk}_{\text{upgrade}})$.	
Other notations	
$\Pr[\mathbb{E}]$	Probability of event $\mathbb{E}$.
$p_{lo/med/hi}$	Discrete probability values: $p_{lo} < p_{med} < p_{hi}$.
τ_i	Time needed to upgrade server i.
t_i	Time when the first i servers have been upgraded.
P_{race}(i)	Probability of mixed-version races at t_i.

Table 1. Summary of notations.

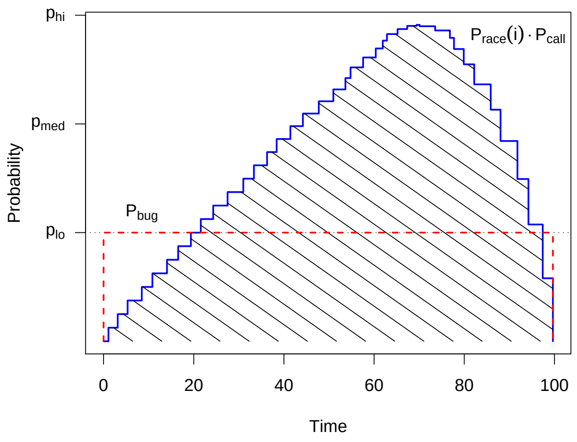
The probability of exposing a bug during normal operation is unaffected by the upgrade process and remains constant: $\Pr[\mathbb{B}_k] = P_{bug}(k) \in \{p_{lo}, p_{med}, p_{hi}\}$. The severity levels $\mathbb{S}(\mathbb{B}_k)$ and $\mathbb{S}(\mathbb{I}_k)$ also remain constant during the rolling upgrade.

The probability of exposing an inconsistency depends on both the workload and the progress of the rolling upgrade. An inconsistency will occur only if the client issues a new callback, which does not exist or has different semantics in the old version (event $\mathbb{E}_1$) and if this callback arrives at a server that has not yet been upgraded and continues to run the old version (event $\mathbb{E}_2$, which corresponds to a mixed-version race). After upgrading the i^{th} server:

$$\begin{aligned}\Pr[\mathbb{I}_k] &= \Pr[\mathbb{I}_k|\mathbb{E}_1] \cdot \Pr[\mathbb{E}_1] = \Pr[\mathbb{E}_2] \cdot \Pr[\mathbb{E}_1] = \\ &= P_{race}(i) \cdot P_{call}(k)\end{aligned}$$



(a) Progression of the rolling upgrade.



(b) The likelihood of triggering an inconsistency, $P_{race}(i) \cdot P_{call}$, varies during the rolling upgrade. The likelihood of exposing a known bug, P_{bug}, remains constant.

Figure 2. Parameters of the risk model.

The probability of mixed-version races P_{race} varies during the upgrade. We note $\tau_1, \tau_2 \ldots \tau_U$ the upgrade durations for servers $1, 2 \ldots U$. The upgrade of the i^{th} server will then be completed at time $t_i = \sum_{k=1}^{i} \tau_k$, as shown in Figure 2a. We do not assume that durations τ_i are known precisely when planning the upgrade. However, we consider that system administrators are able to estimate empirically the expected value of the time needed to upgrade a single host ($\overline{\tau}$), as well as the upper and lower limits (τ_{hi} and τ_{lo}). We use a triangular distribution, characterized by these parameters, to estimate the upgrade timings.

P_{race} depends on two events: sending the initial request to a server running the new version (event $\mathbb{E}_{2.1}$, analogous to step 2 in Figure 1), and sending any of the subsequent callbacks to the old version (event $\mathbb{E}_{2.2}$, analogous to step 4 in Figure 1):

$$\begin{aligned} P_{race}(i) &= \Pr[\mathbb{E}_{2.1} \text{ at } t_i] \cdot \Pr[\mathbb{E}_{2.2} \text{ at } t_i] \\ \Pr[\mathbb{E}_{2.1} \text{ at } t_i] &= \frac{i}{U} \\ \Pr[\mathbb{E}_{2.2} \text{ at } t_i] &= 1 - \Pr[\neg\mathbb{E}_{2.2} \text{ at } t_i] \end{aligned}$$

Event $\neg\mathbb{E}_{2.2}$ corresponds to the scenario where all c callbacks are handled by the new version:

$$\begin{aligned} \Pr[\mathbb{E}_{2.2} \text{ at } t_i] &= 1 - \left(\frac{i}{U}\right)^c \\ P_{race}(i) &= \frac{i}{U} \cdot \left(1 - \left(\frac{i}{U}\right)^c\right) \qquad (1) \end{aligned}$$

$P_{race} = 0$ at times t_0 and t_U, because the first and second terms of the equation are null, respectively. In other words, before and after the rolling upgrade the probability of exposing an inconsistency is 0, because all servers are executing the same version of the software. Figure 2b illustrates the evolution of P_{race} during the rolling upgrade.

We compute the likelihood of exposing bugs or mixed-version inconsistencies by combining the probabilities of the independent events that lead to these circumstances, as shown in Figure 3. After the upgrade of the i^{th} server, the risks of upgrading and of not upgrading are:

$$\begin{aligned} \mathbf{Risk}_{\text{no upgrade}} &= \frac{\sum_{k=1}^{N_{bug}} P_{bug}(k) \cdot \mathbb{S}(\mathbb{B}_k)}{N_{bug} \cdot \max \mathbb{S}} \qquad (2) \\ \mathbf{Risk}_{\text{upgrade}}(i) &= \frac{i}{U} \cdot \left(1 - \left(\frac{i}{U}\right)^c\right) \cdot \\ &\quad \cdot \frac{\sum_{k=1}^{N_{call}} P_{call}(k) \cdot \mathbb{S}(\mathbb{I}_k)}{N_{call} \cdot \max \mathbb{S}} \end{aligned}$$

The risks of upgrading and of not upgrading are functions of the discrete probability values p_{lo}, p_{med}, and p_{hi}. The range of possible risk values is $\mathbf{Risk}_{\mathbb{D}} \in [0, 3p_{lo}]$. We consider that the risk is high when $\mathbf{Risk}_{\mathbb{D}} > 2p_{lo}$, medium when $\mathbf{Risk}_{\mathbb{D}} \in (p_{lo}, 2p_{lo}]$, and low when $\mathbf{Risk}_{\mathbb{D}} \leq p_{lo}$ (see Figure 4).

The average risk of upgrading is:

$$\overline{\mathbf{Risk}_{\text{upgrade}}} = \frac{\sum_{i=1}^{U} \tau_i \cdot \mathbf{Risk}_{\text{upgrade}}(i)}{t_U} \qquad (3)$$

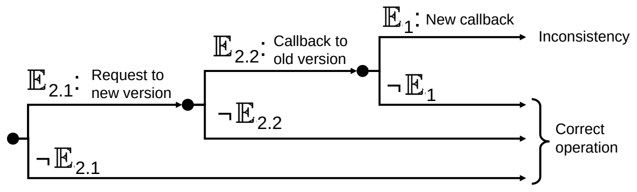


Figure 3. Events leading to a mixed-version inconsistency.

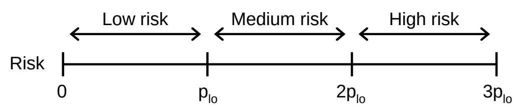


Figure 4. Discrete risk values.

This formula does not have a closed-form expression in terms of $\overline{\tau}$, τ_{lo} and τ_{hi}. Instead, we can estimate this risk through a Monte Carlo simulation, by randomly generating multiple sets of τ_i input terms and by computing the mean of the resulting risks. Using this approach, we can also compute the 95% confidence interval for the average risk of upgrading, which indicates the precision of our estimation.

The maximum risk of upgrading, however, can be computed using a simple, closed-form expression. We compute this maximum by approximating the probability of sending a new callback to the old version, from Equation 1, with a continuous function $\tilde{P}_{race}(x)$ and by differentiating this function:

$$\begin{aligned}\tilde{P}_{race}(x) &= \frac{x}{U} \cdot \left(1 - \left(\frac{x}{U}\right)^c\right) \\ \frac{d\tilde{P}_{race}(x)}{dx} &= 0 \Rightarrow \\ \frac{1}{U} - \frac{(c+1)\cdot x_0^c}{U^{c+1}} &= 0 \Rightarrow \\ x_0 &= U\sqrt[c]{\frac{1}{c+1}}\end{aligned}$$

The maximum probability of sending new callbacks to the old version is:[2]

$$\max(P_{race}) = \sqrt[c]{\frac{1}{c+1}} \cdot \left(1 - \frac{1}{c+1}\right) \quad (4)$$

$\max(P_{race})$ depends only on c, and its asymptotic bound is 1. However, for typical values of c, its value is much lower. If the new version issues up to 12 callbacks into the server, the maximum values of this probability are:

c	1	2	3	4	5	6
$\max(P_{race})$	0.25	0.38	0.47	0.53	0.58	0.62
c	7	8	9	10	11	12
$\max(P_{race})$	0.65	0.68	0.70	0.72	0.73	0.75

The maximum risk of upgrading is:

$$\begin{aligned}\max(\mathbf{Risk}_{\text{upgrade}}) &= \max(P_{race}) \cdot \\ &\cdot \frac{\sum_{k=1}^{N_{call}} P_{call}(k) \cdot \mathbb{S}(\mathbb{I}_k)}{N_{call} \cdot \max \mathbb{S}} \quad (5)\end{aligned}$$

We have created an online tool that automates these calculations, available at `http://orchestrate.cs.vt.edu:8080/examples/servlets/update.html`.

[2] This formula computes an upper bound, because the stair function $P_{race}(x) \leq \tilde{P}_{race}(x)$. However, the exact maximum could be computed by determining the time interval when the risk is maximized, $i = \lfloor x_0 \rfloor$, and introducing it in Equation 1.

4.3 Interpretation

Our risk model compares the expected impacts of executing an upgrade and of putting it on hold. This assessment takes into account the impacts of known bugs in the old version and of mixed-version inconsistencies that can arise during the upgrade. We do not consider the impact of potential bugs in the new version, which cannot be accurately estimated.

The conditional probability of producing an inconsistency, P_{race}, varies as the rolling upgrade progresses. Intuitively, a request that arrives after half of the servers have been upgraded incurs a higher risk of inconsistency than requests arriving at the beginning or at the end of the upgrade. Therefore, the decision whether to upgrade or not can take into account either the maximum or the average risk over the duration of the rolling upgrade. Most system administrator will base this decision on the average risk, which corresponds to the intuitive notion of expected impact of the upgrade. However, mission-critical systems, where each request can have a severe impact (e.g., physical injury or financial loss), will consider the maximum risk of upgrading.

While we consider that P_{bug} and P_{call} remain constant for the duration of the upgrade, these parameters are likely to be dependent on the system's workload. For example, on different days of the week the load might shift between different features provided by the system, exercising different code paths in the old and new software versions. This will change the probabilities of exposing bugs and inconsistencies. If the system administrators can estimate the values for P_{bug} and P_{call} during different time windows, based on testing results and knowledge of past workloads, our model will help them determine the best time for performing the upgrade. Alternatively, the risk assessment may suggest that an offline upgrade, executed during a planned maintenance window, is more appropriate for the system.

5. Model validation

Complete data on real-world upgrade failures is scarce and hard to obtain, due to the sensitivity of this subject. We are aware of two real-world examples of upgrade failures that can be traced to mixed-version races [12, 23]. Because, to the best of our knowledge, this race condition has not been characterized before, we currently lack sufficient data to design statistically-significant experiments for evaluating the risk of upgrading in the presence of mixed-version races. Moreover, our analytical model assesses the perceived impact of upgrades, which cannot be measured directly. In particular, the severity of a bug or of a mixed-version inconsistency is a qualitative measure that reflects the developers' or administrators' perception of the impact resulting from the manifestation of these bugs/inconsistencies. This *a priori* perception of impact is difficult to correlate with a measurable quantity.

We conduct a qualitative evaluation of our risk model, seeking to answer the question: *Is this risk model use-*

ful? By walking through three hypothetical—but realistic—scenarios of online upgrades, we focus on the time when a system administrator must decide whether to upgrade or not and on the information available for making this decision. Two scenarios focus on mission-critical systems (online banking, in Section 5.1, and foreign exchange, in Section 5.3) and one focuses on a large-scale system that is not mission critical (a social networking site, in Section 5.2). We show that using our analytical model leads to better decisions than those suggested by intuition alone. These scenarios demonstrate that the model provides additional information, not available through other means, for making the upgrade-or-not decision. Our risk model can systematically inform an upgrade administrator, or any other stakeholders in these applications, whether an online upgrade is appropriate in their environment.

In this paper, we do not seek to answer the question: *How accurate is the additional information provided by the risk model?* In the future, we plan to use the model in a production system, for an extended period of time, and to report on this experience after observing real upgrade failures. We believe that such practical experience is essential for providing a complete validation of our proposed approach.

5.1 Upgrade #1: Online banking

Imagine that a bug in the Web interface of an online banking application (such as the one described in Section 3) was reported and corrected. Specifically, in the old version, an edit box for entering fund transfer information accepts all alpha-numeric characters rather than restricting user input to numbers only. The alphanumeric characters are needed in order to enter a currency specification. However, this can expose the site to a SQL injection attack, which is one of the top 25 programming errors that lead to security vulnerabilities [8]. The new version of the Web interface uses a radio box to specify the currency and a numbers-only text box. Because this bug afflicts those users that use online brokerage services, who tend to constitute an important segment of the customers, the bug is assigned the severity level 5 (highest).

Through integration testing, it has been determined that replacing the upgrade can lead to an inconsistency resulting from a mixed-version race. Because the old version of the server-side code expects a single parameter, it will disregard the currency specification and will assume that the sum is specified in US dollars. This can cause significant problems when the site is used by customers with accounts in foreign currencies. This potential inconsistency is assigned severity level 3.

Because the impact of SQL injection attacks outweighs the severity of mixed-version inconsistencies, intuition suggests that the upgrade should be deployed as soon as possible. However, the most likely impacts of these two events depend on other parameters as well. Imagine that the probability of being the target of an attack is $P_{bug} = p_{lo}$, while

	§5.1	§5.2	§5.3
U	10	100	100
$\overline{\tau}$	1 min	1 min	2 min
τ_{lo}	0 min	0 min	0 min
τ_{hi}	6 min	2 min	7 min
P_{call}	p_{hi}	p_{hi}	p_{med}
c	6	2	1
$\mathbb{S}(\mathbb{I})$	3	5	3
P_{bug}	p_{lo}	p_{med}	p_{hi}
$\mathbb{S}(\mathbb{B})$	5	5	2
$\mathbf{Risk}_{\text{no upgrade}}$	Low	Medium	Medium
$\max(\mathbf{Risk}_{\text{upgrade}})$	Medium	Medium	Low
$\overline{\mathbf{Risk}}_{\text{upgrade}}$	–	Low	Low

Table 2. Risk parameters in the three upgrade scenarios.

most of the callbacks issued by the new version use the new radio box parameter ($P_{call} = p_{hi}$), because the majority of the bank's customers have accounts in a foreign currency (the remaining parameters are summarized in Table 2).

Using our online tool, we compute that the risk of not upgrading is low, while the maximum risk of upgrading is medium. Because online banking is a mission-critical application, we do not take the mean risk of upgrading into consideration. Contrary to our intuition, the analytical model predicts that it is better to upgrade during a planned maintenance window than online. Alternatively, an online upgrade may be appropriate during a time window when most of the customers who access the system have accounts in dollars.

5.2 Upgrade #2: Social networking site

The Web interface of a social-networking site is not rendered correctly when accessed using an old version of some Web browser. Specifically, a push button that allows users to log in appears disabled. This happens because the browser in question uses an obsolete version of the DOM tree. The usage monitoring service in place indicates that a user will try to access the Web site using this particular version of the browser with probability $P_{bug} = p_{med}$. However, the bug is assigned severity level 5 because it causes the site to be unavailable whenever it occurs, and high availability is a top priority for the social networking site.

After log-in, the old version of the server sends (via AJAX callbacks) more information than the user needs. The client-side code, running in the user's browser, filters this information. The new version, which fixes the DOM bug, changes the way elements are displayed and moves the filtering to the server side. Whenever a new-version callback is processed by an old-version server, some other user's private information is leaked and displayed in the browser ($P_{call} = p_{hi}$). This potential privacy breach is also assigned severity level 5.

Our intuition suggests that an online upgrade should be avoided, because, while the bug and the mixed-version inconsistency are equally severe, the bug does not manifest frequently. However, as social networking is not a mission critical application, we compare the risk of not upgrading (medium) with the average risk of upgrading (low). In this case, the analytical risk model predicts that an online upgrade represents the best course of action.

5.3 Upgrade #3: Foreign exchange system

Multiple online banking applications rely on a cloud-based service that provides foreign-currency exchange rates. This cloud-based service is provisioned and upgraded by a third party. The cloud service can support multiple versions of the communication protocol, and the version in use is established at the start of the message exchange. The service uses a publish-subscribe infrastructure. When banking applications subscribe to the service, they receive asynchronous messages that encapsulate Java objects. The new version of the service is provided as an extension of the old service; the corresponding objects instantiate a subclass of the old version's data type.

A certain bank requires the new version of the service in order to provide a new feature. Specifically, in addition to the current exchange rate, the new version also specifies the time when this rate was valid. This information is useful for customers who engage in money market speculation. This missing feature is assigned severity level 2. A sizable subsegment of the system's users, are estimated to wish the feature added ($P_{bug} = p_{hi}$).

However, an online upgrade can expose a mixed-version race. If the bank starts a rolling upgrade, to add the new feature in its application code, the service publisher will begin broadcasting messages belonging to the new version. Some messages will be received by servers still running the old version ($P_{call} = p_{med}$). When these servers unmarshall the message and determine that the object's class definition is unknown, they will throw an exception. This renders the service unavailable for servers that have not yet been upgraded. This partial outage is assigned severity level 3.

The missing feature and the partial outage have different likelihoods and different severity levels. It is, therefore, difficult to make a decision based only on intuition. Our online tool shows that the risk of upgrading is always lower than the risk of not upgrading, and recommends an online upgrade. Our analytical model provides a systematic approach for deciding whether to upgrade or not to upgrade.

6. Discussion

Our risk model can determine, analytically, the best time window for performing an upgrade. Anecdotal evidence, and recent empirical studies, indeed suggest that some days might be better than others for implementing changes and upgrades. For example, Śliwerski et al. [27] study the version history of several open-source systems and discover a temporal correlation between the code changes that require subsequent fixes and the weekday when these changes are implemented. According to this study, the best days for fixing software bugs are Tuesdays (with Fridays and Saturdays being the riskiest days). Interestingly, Windows [17] and Facebook [23] also deploy their upgrades on Tuesdays.

Recent advances in low-overhead dynamic analysis [4, 15] have made it possible to monitor systems in their deployment environments in order to assess the probability that certain bugs will be exposed. These techniques provide the tools for evaluating the risk of not upgrading a system that includes known software defects ($\mathbf{Risk}_{\text{no upgrade}}$).

However, the leading cause of failure for enterprise upgrades are errors in the upgrade procedure (e.g., specifying wrong service locations, introducing shared-library conflicts, creating database-schema mismatches), rather than software defects (e.g., bugs in the new version) [7, 10]. Moreover, these failures are often hard to replicate outside of the deployment environment, because they correspond to broken dependencies and their manifestation is workload-dependent. Unlike for software defects, we currently lack a comprehensive corpus of realistic faults that commonly occur during online upgrades and the conditions that trigger them. This makes it challenging to assess the terms of the risk of upgrading ($\mathbf{Risk}_{\text{upgrade}}$), such as the severity of mixed-version inconsistencies. In the future, we plan to establish a collaborative repository of data on upgrade failures, collected from multiple industry sources. Similar repositories, such as the common programming errors that lead to security vulnerabilities [8], have had a great impact on the practice of programming, and our paper emphasizes the utility of an upgrade-centric fault repository.

During enterprise-system upgrades, the mixed-version states are usually an artifact of the upgrade approach (e.g., rolling upgrades). However, sometimes mixed versions represent a user requirement. For example, when the operators of an enterprise system must undergo an extensive retraining to use the new version of system, the upgrade must be deployed gradually [9]. These user requirements add to the complexity of impact assessment for online upgrades, by extending the time when the system is vulnerable to mixed version races and by increasing the number of program versions that co-exist in the system.

Mixed-version races are an example of behavior that emerges at runtime and that cannot be tested, exhaustively, before the system is deployed. Large-scale systems that undergo runtime evolution (e.g., online software-upgrades, architectural reconfigurations) must cope with changes implemented during the system execution. These changes interact with the workload in ways that may be unpredictable at design-time. Reasoning about such emerging behavior is difficult because previously-established system invariants do not hold, changes are implemented by both human and software agents, hidden dependencies in the environment can induce upgrade failures, and externally-imposed deadlines might affect the outcome. This paper represents a first step toward a systematic approach for validating such runtime-emerging behaviors.

7. Conclusion

We describe a new type of race condition that may occur during online upgrades in systems spanning multiple administrative domains and communicating via asynchronous messaging across domain boundaries. The recorded occurrences of such mixed-version races suggest that they can produce severe effects, including financial loss. Mixed-version races will become widespread in systems relying on cloud-computing resources, which are provisioned and operated by third-party service providers. We introduce an analytical model and an online tool for comparing the risks of upgrading and of not upgrading. This model compares the expected impact of mixed version races with the effects of known bugs in the deployed software. Our model represents a first step toward reasoning about the behavior of system states that emerge in the deployment environment and that may be unpredictable at design-time.

References

[1] S. Ajmani, B. Liskov, and L. Shrira. Modular software upgrades for distributed systems. In *European Conference on Object-Oriented Programming*, pages 452–476, Nantes, France, Jul 2006.

[2] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright. Timing the application of security patches for optimal uptime. In *Large Installation System Administration Conference*, pages 233–242, Philadelphia, PA, Nov 2002.

[3] T. Bloom. *Dynamic Module Replacement in a Distributed Programming System*. PhD thesis, MIT, 1983.

[4] M. Bond, K. Coons, and K. McKinley. Pacer: Proportional detection of data races. In *ACM Conference on Programming Language Design and Implementation*, Toronto, CA, Jun 2010.

[5] E. A. Brewer. Lessons from giant-scale services. *IEEE Internet Computing*, 5(4):46–55, Jul/Aug 2001.

[6] A. Choi. Online application upgrade using edition-based redefinition. In *ACM Workshop on Hot Topics in Software Upgrades*, Orlando, FL, Oct 2009.

[7] O. Crameri, N. Knežević, D. Kostić, R. Bianchini, and W. Zwaenepoel. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Symposium on Operating Systems Principles*, pages 221–236, Stevenson, WA, Oct 2007.

[8] CWE/SANS. Top 25 most dangerous programming errors. Feb 2010.

[9] A. Downing, Oracle Corporation. Personal communication, 2008.

[10] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it? Toward dependable, online upgrades in enterprise systems. In *ACM/IEEE/IFIP Middleware Conference*, pages 349–372, Urbana-Champaign, IL, Nov/Dec 2009.

[11] T. Dumitraş, D. Roşu, A. Dan, and P. Narasimhan. Ecotopia: An ecological framework for change management in distributed systems. In C. Gacek, A. Romanovsky, and R. de Lemos, editors, *Architecting Dependable Systems IV*, pages 262–286. Springer-Verlag, LNCS 4615, 2007.

[12] S. Hansell. Glitch makes teller machines take twice what they give. *The New York Times*, Feb 18 1994.

[13] M. Hicks. *Dynamic Software Updating*. PhD thesis, Department of Computer and Information Science, University of Pennsylvania, August 2001.

[14] J. Kramer and J. Magee. Dynamic configuration for distributed systems. *IEEE Transactions on Software Engineering*, 11(4):424–436, 1985.

[15] B. Liblit, A. Aiken, A. X. Zheng, and M. I. Jordan. Bug isolation via remote program sampling. In *ACM Conference on Programming Language Design and Implementation*, San Diego, CA, Jun 2003.

[16] Microsoft Corporation. Perform a rolling upgrade from Windows 2000. TechNet Library, Jan 2005. `http://technet.microsoft.com/en-us/library/cc738005(WS.10).aspx`.

[17] Microsoft Developer Network. *Windows Update Agent*. `http://msdn2.microsoft.com/en-us/library/aa387099.aspx`. Retrieved on 18 Feb 2008.

[18] Office of Government Commerce. *Service Transition*. Information Technology Infrastructure Library (ITIL). 2007.

[19] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and validating database system administration. *USENIX Annual Technical Conference*, Jun 2006.

[20] D. Oppenheimer, A. Ganapathi, and D. A. Patterson. Why do Internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, Mar 2003.

[21] Oracle Corporation. Database rolling upgrade using Data Guard SQL Apply. Maximum Availability Architecture White Paper, Dec 2008. `http://www.oracle.com/technology/deploy/availability/pdf/maa_wp_10gr2_rollingupgradebestpractices.pdf`.

[22] D. Patterson. A simple way to estimate the cost of downtime. In *Large Installation System Administration Conference*, pages 185–188, Philadelphia, PA, Nov 2002.

[23] D. Reiss, Facebook. Personal communication, 2009.

[24] J. S. Rellermeyer, M. Duller, and G. Alonso. Consistently applying updates to compositions of distributed OSGi modules. In *ACM Workshop on Hot Topics in Software Upgrades*, Nashville, Tennessee, Oct 2008.

[25] M. Segal. Online software upgrading: new research directions and practical considerations. In *Computer Software and Applications Conference*, pages 977–981, Oxford, England, Aug 2002.

[26] M. E. Segal and O. Frieder. Dynamically updating distributed software: supporting change in uncertain and mistrustful environments. In *IEEE Conference on Software Maintenance*, pages 254–261, Oct 1989.

[27] J. Śliwerski, T. Zimmermann, and A. Zeller. When do changes induce fixes? On Fridays. In *International Workshop on Mining Software Repositories (MSR)*, Saint Louis, Missouri, May 2005.

[28] E. B. Swanson. The dimensions of maintenance. In *International Conference on Software Engineering*, pages 492–497, San Francisco, CA, 1976.

[29] L. Tewksbury, L. Moser, and M. Melliar-Smith. Live upgrades of CORBA applications using object replication. In *International Conference on Software Maintenance*, pages 488–497, Florence, Italy, Nov 2001.

[30] S. Vinoski. Convenience over correctness. *IEEE Internet Computing*, 12(4):89–92, 2008.

[31] W. Zheng, R. Bianchini, G. J. Janakiraman, J. R. Santos, and Y. Turner. Justrunit: Experiment-based management of virtualized data centers. In *USENIX Annual Technical Conference*, San Diego, CA, Jun 2009.