Carnegie Mellon University

CARNEGIE INSTITUTE OF TECHNOLOGY

THESIS

SUBMITTED IN PARTIAL FULFILLMENT OF THE REQUIREMENTS

FOR THE DEGREE OF Doctor of Philosophy

TITLE Improving the Dependability of Distributed Systems through AIR Software Upgrades

PRESENTED BY Tudor A. Dumitraş

ACCEPTED BY THE DEPARTMENT OF

Electrical & Computer Engineering

_____ _____

ADVISOR, MAJOR PROFESSOR DATE

_____ _____

DEPARTMENT HEAD DATE

APPROVED BY THE COLLEGE COUNCIL

_____ _____

DEAN DATE

Improving the Dependability of Distributed Systems through AIR Software Upgrades

Submitted in partial fulfillment of the requirements for

the degree of

Doctor of Philosophy

in

Electrical & Computer Engineering

Tudor A. Dumitraş

B.S., Computer Science, “Politehnica” University, Bucharest, Romania
Diplôme d’Ingénieur, Ecole Polytechnique, Paris, France
M.S., Electrical & Computer Engineering, Carnegie Mellon University

Carnegie Mellon University
Pittsburgh, PA

December, 2010

To my parents and my teachers, who showed me the way. To my friends, who gave me a place to stand on. Pentru Tanti Lola.

Abstract

Traditional fault-tolerance mechanisms concentrate almost entirely on responding to, avoiding, or tolerating unexpected faults or security violations. However, scheduled events, such as *software upgrades*, account for most of the system unavailability and often introduce data corruption or latent errors. Through two empirical studies, this dissertation identifies the *leading causes of upgrade failure*—breaking hidden dependencies—*and of planned downtime*—complex data conversions—in distributed enterprise systems. These findings represent the foundation of a new benchmark for software-upgrade dependability.

This dissertation further introduces the *AIR properties*—Atomicity, Isolation and Runtime-testing—required for improving the dependability of distributed systems that undergo major software upgrades. The AIR properties are realized in *Imago*, a system designed to reduce both planned and unplanned downtime by upgrading distributed systems end-to-end. Imago builds upon the idea of isolating the production system from the upgrade operations, in order to avoid breaking hidden dependencies and to decouple the data conversions from the normal system operation. Imago includes novel mechanisms, such as providing a parallel universe for the new version, performing data conversions opportunistically, intercepting the live workload at the ingress and egress points or executing an atomic switchover to the new version, which allow it to deliver the AIR properties.

Imago harnesses opportunities provided by the emerging cloud-computing technologies, by trading resource overhead (needed by the parallel universe) for an improved dependability of the software upgrades. This approach separates the functional aspects of the upgrade from the mechanisms for online upgrade, enabling an *upgrade-as-a-service* model. This dissertation also describes techniques for assessing the impact of software upgrades, in order to reason about the implications of *relaxing the AIR guarantees*.

This is for the one that I ate, 32 years ago.

Acknowledgments

Since I started to learn about Computer Science, I have been fascinated by our ability to write computer programs that can protect themselves from various adverse conditions in their environments. I equally enjoy building dependable systems firsthand and studying empirically the behavior of systems large and small. This is the result of my interactions with a few great mentors, who helped me discover the secrets of Computer Science and who showed me the ways of scientific research.

At age 13, I received a great gift from my uncle, Florin Covaciu. It was an HC-85 computer, a Romanian replica of the Sinclair ZX Spectrum (one of the world's first personal computers). I learned to program on this machine, and, since then, I was not able to stay away from programming for too long. This path took me to the Computer-Science High School in Bucharest (*Liceul de Informatică*, now *Colegiul Național Tudor Vianu*), where teachers Mihai Budiu and Raluca Vasilescu opened my eyes to many computing concepts and to their power to transform human society. These teachers shared their great knowledge and passion for computing, and this inspired me to continue my studies in this field. I was not too surprised when I met Mihai and Raluca again, years later, in the graduate program at Carnegie Mellon University.

Among my professors at the "Politehnica" University in Bucharest were Mircea Petrescu, who had supervised my father's Honors thesis (*diploma de licență*) twenty-five years previously, Adrian Petrescu and Francisc Iacob, the creators of the HC-85, and Nicolae Țăpuș, who supervised my own Honors thesis on Argo, a search engine with a distributed Web crawler. Most of all, I am grateful to Zoea Racoviță for encouraging me to continue my graduate studies and to apply to the Ph.D. program at Carnegie Mellon University.

My education at the Ecole Polytechnique in Paris taught me to be responsible, confident, determined and to keep my commitments. I have to thank Jean-Marc Steyaert for guiding me through those years. Atom, a final-year project supervised by Sam Toueg, fo-

cused on implementing a practical group-communication system based on unreliable failure detectors (an exclusively theoretical technique, at the time) and seeded my curiosity about distributed systems.

When I arrived at Carnegie Mellon, Radu Mărculescu taught me the paramount importance of originality in all academic endeavors and the value of anticipating future technology trends for systems-oriented research. Our first paper together, which identified the need for system-level fault tolerance in the emerging networks-on-chip (NoC) and reevaluated fundamental trade-offs made by classical networking protocols, remains my most cited publication. Phil Koopman taught me everything I know about dependability, and he always gave me good advice throughout my graduate studies.

Many people contributed to the approach described in this dissertation. The members of my thesis committee, Greg Ganger, Bruce Maggs and Asit Dan, encouraged me to pursue the topic of online software upgrades and provided invaluable feedback in all the stages of this research. Dan Siewiorek taught me how to create a rigorous taxonomy. Jiaqi Tan and Zhengheng Gho helped me design some of the core algorithms of Imago. At my request, Lorenzo Keller wrote a user manual for ConfErr, his fault-injection tool for mutating configuration files. A dinner-time discussion with Eli Tilevich turned into a publication about the risks of software upgrades across multiple administrative domains. Douglas Schmidt showed me how to present my research in an effective way. Jean-Charles Fabre has always been a great mentor and a true friend. *Je n'oublierai jamais que je dois mon premier emploi à ton soutien sans réserve, Jean-Charles.*

Daniela Roșu, Bich Le and Alan Downing—my internship supervisors at IBM Research, VMware and Oracle, respectively—provided me with a wealth of information about the practical challenges of performing software upgrades. My approach also incorporates extensive feedback from the industry members of the Parallel Data Lab consortium. During my dissertation research, I received financial support from the NSF CAREER Award CCR-0238381, the DARPA PCES contract F33615-03-C-4110, as well as Carnegie Mellon's CyLab and Parallel Data Lab.

While not directly involved in this research, my collaborators Danny Dig and Iulian Neamtiu helped me establish the series of workshops on Hot Topics in Software Upgrades (HotSWUp). The first two editions of the workshop provided a venue for insightful discussions about how upgrades are performed at various levels—in the front-end of cloud com-

puting infrastructures, in EJB-based enterprise applications, in databases, in long-running servers, in middleware frameworks or in satellites orbiting the Earth. Most importantly, HotSWUp emphasized that the challenge of upgrading distributed systems end-to-end calls for an inter-disciplinary approach, combining ideas and techniques from several areas of Computer Science.

Above all, I would like to thank my dissertation advisor, Prof. Priya Narasimhan, for all her guidance. She taught me the secrets of fault-tolerant middleware, and she showed me how to enhance legacy applications with replication and recovery mechanisms by transparently intercepting the application's system calls. She shared with me her great gift for writing and presenting technical material, she taught me the scientific method and she showed me how to ask the questions that matter. She passed down the teachings of Gottfried Wilhelm Leibniz, our academic ancestor, about how to turn an abundance of experimental data into rigorous scientific findings. She also passed down her aversion of adjectives and hyperbolae. She transformed a student into a researcher.

Outside of Computer Science, my good friends—too many to list here—have always been a source of inspiration and vitality. My aunt, Ștefania Dumitraș, taught me about the practical things in life, and she also taught me French. My cousin, Ioana Căprar, has been close as a sister. My parents, Dan Dumitraș and Monica Dumitraș, are my ultimate role models. Working at the Institute of Atomic Physics (where the first Romanian computer, CIFA, was built in 1955), their group of friends consisted of many other researchers, recognized internationally. I grew up among idealistic and passionate people, who were pushing the limits of science and engineering, and this has influenced who I am today. I am grateful for all the intellectual gifts you gave me. I also thank Corina, with all my heart, for her support and encouragement during the final stages of my dissertation writing. *Ești un peștișor de aur.*

This dissertation is the fruit of your labor, as much as mine's. For my part, this experience helped me to grow up, professionally, and to understand the difference between good research and great research. I continue to marvel at the beauty of Computer Science and at all the things that are out there, for us to discover.

Contents

List of Figures

List of Tables

> We hear desperate cries for a silver bullet—something to make software costs drop as rapidly as computer hardware costs do.
>
> F. Brooks, No silver bullet, 1987

Chapter 1

Introduction

MODERN distributed systems are perhaps the most intricate structures ever engineered, and their benefits for society are impaired by our inability to make end-to-end dependability guarantees. Software dependability remains challenging despite recent advances in preventing and finding software bugs, or improvements in unit and integration testing. While these techniques reduce the complexity of an *accidental* task—the need to express conceptual specifications in a programming language—managing change in software systems represents one of the *essential* obstacles for their dependability [Brooks, 1987].

Even after their deployment in the field, successful distributed systems are expected to change frequently, in order to add new features, to improve performance and scalability, to conform with government regulations or to reduce operating costs by switching software vendors. Unlike hardware or mechanical systems, computer programs can be modified with relative ease. However, when deploying these changes in an actively-used system, through software upgrades, we must preserve the ecosystem of dependencies from the operational environment. For this reason, the question *How to perform software upgrades dependably?* represents a grand challenge for distributed-systems research [Kaashoek et al., 2005].

Traditional approaches for ensuring dependability [Avižienis et al., 2004] concentrate almost entirely on responding to, avoiding, or tolerating unexpected faults or security violations. However, intentional software changes, such as software upgrades, account for 66%–86% of the time when the service is not available, reportedly (see Table 1.1). The need for such *planned downtime* stems from the current limitations of upgrade mechanisms, which are unable to upgrade distributed systems atomically, end-to-end. Furthermore, software

Table 1.1. Comparison of studies of distributed-system availability. Software changes account for most of the planned downtime. The contribution of change-management errors (*e.g.* software upgrades) to unplanned downtime has increased during the past two decades.

Unplanned Downtime	
[Gray, 1990]	*27.3% of unplanned downtime* due to errors in operations procedures or in system configurations, in Tandem systems.
[Oppenheimer et al., 2003]	*75% of unplanned downtime* due to operations errors during change-management procedures, in Internet services.
Planned Downtime	
[Lowell et al., 2004]	*86% of total downtime* due to planned software maintenance, in high-availability applications.
[Malik and Scott, 2008]	*66%–76% of total downtime* due to planned software maintenance, in enterprise systems.

upgrades cause system failures, such as *unplanned downtime*, partial outages, latent errors or data corruption. Recent studies suggest that up to half of software upgrades fail [Crameri et al., 2007], and that these failures account for 75% of the unplanned downtime [Oppenheimer et al., 2003].

As the instruments of economic activity turn to the Internet, enterprises can no longer afford to incur such planned and unplanned downtime and must perform software upgrades online, without stopping their systems. Historically, mechanisms for *online upgrade* were developed for the telecommunications industry [for example, in AT&T's 5ESS switch: Toy, 1992]. These mechanisms focus on updating single-node systems on the fly, without stopping the running program. Industry trends suggest, however, that online upgrades are currently needed in *large-scale distributed systems*, such as electrical utilities, assembly-line manufacturing, customer support, e-commerce or online banking [Choi, 2009]. 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 depend on more complex interactions among the heterogeneous system components (*e.g.*, asynchronous messaging, long-running transactions, reads/writes to shared storage). In distributed systems spanning multiple administrative domains, it may be difficult to coor-

dinate the operations performed during an online upgrade. Moreover, in some distributed architectures most of the system functionality depends on a single, shared component (*e.g.*, the database), which cannot be upgraded without changing the rest of the system as well, effectively preventing partial or gradual upgrades.

This dissertation identifies the leading causes of upgrade failure—breaking hidden dependencies—and of upgrade-induced planned downtime—database schema evolution. Building on these empirically-derived insights, this dissertation explores the idea of isolating a live production system from the upgrade operations, with the aim of preventing the upgrade-specific faults from breaking hidden dependencies.

I take a holistic approach and focus on upgrading distributed systems end-to-end. The full distributed-system upgrade is an atomic operation, executed online even when performing complex schema and data conversions. I present Imago[1], an upgrading system that harnesses the opportunities provided by the emerging cloud computing technologies to simplify large-scale upgrades, to allow upgrades to be executed efficiently online, and to improve their dependability. This approach separates the functional aspects of the upgrade (*e.g.* persistent-data conversion) from the mechanisms for upgrading online (*e.g.* atomic switchover) and enables an upgrade-as-a-service model.

1.1 The dependability of software upgrades

System dependability has several attributes, such as availability and reliability [Avižienis et al., 2004]. Availability is the fraction of time that a system is ready to provide correct service and does not experience planned or unplanned downtime. Reliability is a time-dependent function expressing the probability that the system will provide correct service. A large body of anecdotal evidence suggests that, in practice, software upgrades are unreliable and often cause downtime, latent errors or data corruption. For example, in November 2003, the upgrade of a customer-relationship management system at AT&T Wireless backfired, causing chronic downtime in several key systems and affecting 50,000 customers per week [Koch, 2004]. The complexity of dependencies on 15 legacy back-end systems was unmanageable, and the integration could not be tested in a realistic environment. The up-

[1]The *imago* is the final stage of an insect or animal that undergoes a metamorphosis, *e.g.*, a butterfly after emerging from the chrysalis [Oxford English Dictionary, 1989].

grade caused repeated crashes and a ripple effect that disabled other AT&T systems as well. Rollback was impossible because not enough of the old system had been preserved. The negative effects lasted for 3 months, and the company lost $100 million in revenue.

Some dependencies, such as the ways that persistent data objects or observed performance levels can affect system behavior, are particularly hard to detect. In February and September 2009, Google's web-based email service, Gmail, experienced two outages, each causing around 2 hours of downtime, following routine software upgrades [Cruz, 2009; Treynor, 2009]. The new functionality—which had been designed to improve service availability—resulted in additional load that overwhelmed the servers in other data centers. Moreover, breaking such non-functional dependencies during an upgrade can cause failures that persist even after rolling back the changes. For example, in August 1996, an upgrade in the main data center of America Online (AOL)—the world's largest Internet Service Provider at the time—was followed by a 19-hour outage [Neumann et al., 1996]. The system behavior did not improve even after the upgrade was rolled back because the routing tables had been corrupted during the upgrade. These examples illustrate that the impact of changes is often difficult to predict before performing a software upgrade.

Upgrade failures affect safety-critical systems as well. In March 2008, the upgrade of an enterprise system used for business analytics forced a nuclear power plant into a 48-hour emergency shutdown [Krebs, 2008]. The system administrator who installed the upgrade was not aware that the software was designed to synchronize data with the plant's primary control system. The upgrade re-initialized the chemical and diagnostic data on both systems, and this caused a perceived drop in the nuclear coolant levels that automatically triggered an emergency shutdown. In 2006, following a network upgrade, an automated drug dispenser went offline in the emergency room of a hospital [Wears et al., 2006]. While the system upgraded was not time sensitive, the failure prevented a patient in critical condition from receiving the appropriate medication.

The upgrade failures described above had different root causes and affected different system components. In all these examples, however, the failure of an upgrade in one system affected other, apparently unrelated, systems of the enterprise.

1.1.1 Upgrading inter-dependent systems

Existing upgrade techniques rely on tracking the complex dependencies among the distributed system components. When the old and new versions of the system-under-upgrade share dependencies (*e.g.*, they rely on the same third-party component but require different versions of its API), the upgrade procedure must avoid breaking these dependencies in order to prevent unplanned downtime or data-loss.

For single-host systems, the effects of broken dependencies are known, colloquially, as "DLL Hell": when installing or upgrading an application together with all of the libraries that it depends upon, other unrelated applications might be inadvertently disrupted or rendered inoperable because a shared library (the dependency) was removed or replaced with an incompatible version [Anderson, 2000]. In some cases, applications might depend on a specific version of a shared library, because an older version might not be able to provide the required functionality and newer versions can introduce breaking API changes [Dig and Johnson, 2006].

To prevent breaking these dependencies during an upgrade, modern operating systems provide package managers that determine automatically how to install a new package, or upgrade an existing one, along with all of its dependencies. These tools include APT [Silva, 2005] for Debian Linux, YUM [Brown and Pickard, 2003] for RedHat Linux, Portage [Vermeulen et al., 2007] for Gentoo Linux and the Windows Update Agent [Leyden, 2003] for Microsoft Windows. Package-management systems rely on dependency-tracking by maintaining repositories of packaged software components, along with metadata that describes the dependency and conflict relationships among all the packages that an instance of the corresponding operating system may need. Figure 1.1 shows the dependencies, obtained from the APT package manager, among the software components from a single host of a typical enterprise system. This host includes an Apache web server that loads the PHP interpreter and a client library for the MySQL database server, which depend on a complex graph of third-party components.

In practice, dependencies are often poorly documented [Dig and Johnson, 2006; Egyed, 2003], and they cannot always be detected automatically. Dependency-discovery techniques, such as static analysis of object code [Sun and Couch, 2001], of source code [Dig et al., 2006] or of configuration files [Magoutis et al., 2008], semantic analysis [Dig et al.,

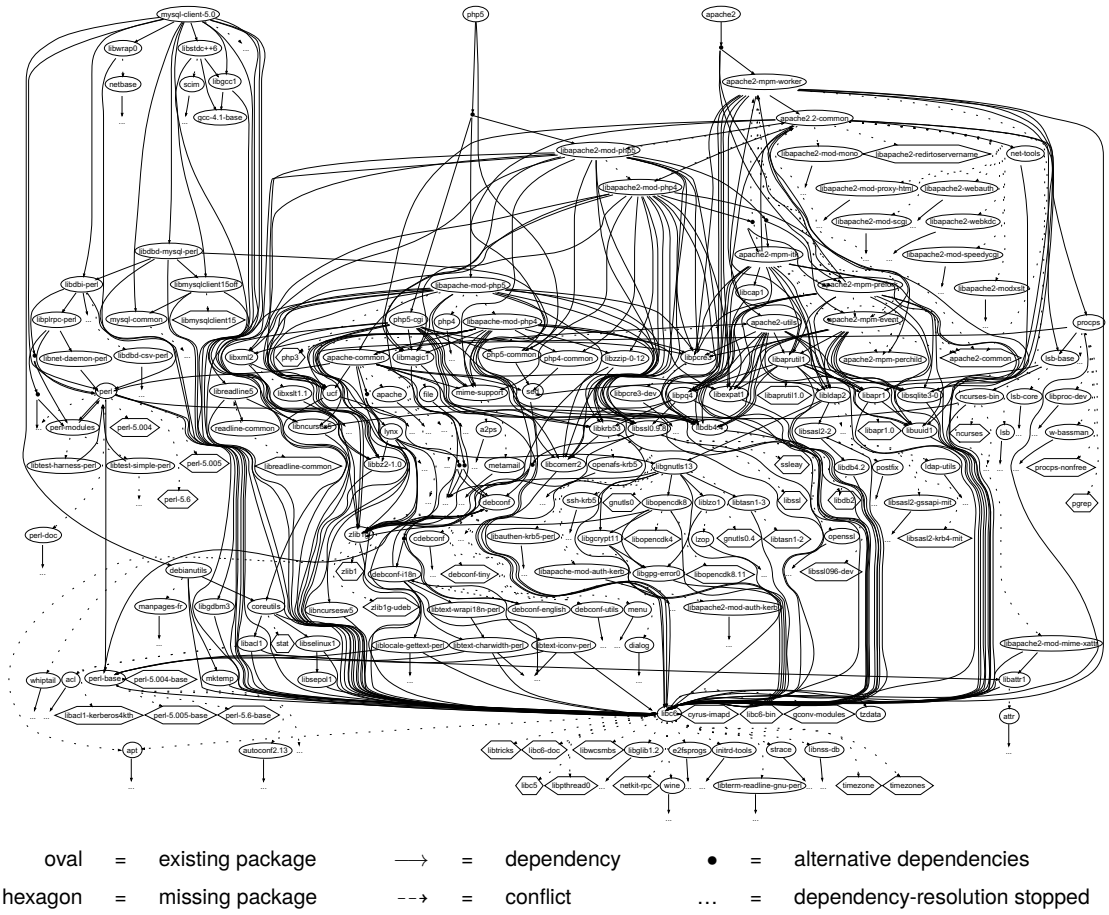


Figure 1.1. Example of dependencies in a single host. The functionality of a typical node from an enterprise system depends on a complex graph of third-party libraries.

2006], runtime monitoring [Dunagan et al., 2004] or active perturbation [Brown et al., 2001] cannot provide a complete coverage of all the factors that might influence the behavior of a distributed system. For example, the complete set of shared libraries that might be loaded by an application cannot be discovered using static analysis (*e.g.*, if the libraries are specified through dependency injection techniques [Fowler, 2004]), and monitoring the library loading operations at runtime is a best-effort approach that might not cover all of the possible application behaviors. Dependency tracking ultimately relies on metadata that is *manually maintained* by teams of developers and quality-assurance engineers through a time-intensive and error-prone process.

Moreover, determining the correct configuration of an upgraded system by resolving the dependencies of the installed components is an *NP-complete* problem (see Appendix A for the proof). Existing approaches for tracking dependencies use heuristics [for example:

Silva, 2005; Brown and Pickard, 2003] to ensure that the search for a correct configuration terminates in a timely fashion, or they rely on SAT solvers [Tucker et al., 2007; Di Cosmo, 2005] to guarantee that a solution will be found for all installable packages. Heuristics-based approaches might fail to find a solution where one exists, while, for certain corner cases, the run time of SAT solvers is exponential in the size of the repository [Tucker et al., 2007].

This suggests that the size of dependency repositories will determine the point where ensuring the correctness of upgrades through dependency tracking becomes computationally infeasible. Current repositories contain approximately 25,000 packages [Di Cosmo et al., 2008]. If dependencies on configuration settings are included, the size of these repositories would increase by an order of magnitude (a typical instance of the Windows XP operating system has 200,000 configuration settings that may be shared by several applications [Wang et al., 2003]). Moreover, the behavior of a distributed system depends on heterogeneous third-party components, APIs, data objects, communication protocols, Internet routes or performance levels. The graph from Figure 1.1 represents but a small fraction of the complex dependency relationships among the components of a real-world distributed-system.

Because dependencies in distributed systems are increasingly complex and because runtime dependencies cannot always be discovered automatically, the benefits of dependency tracking are reaching their limit. The current effect of these fundamental limitations is that sometimes *dependencies remain hidden* from the administrators who coordinate system-wide software upgrades. This can cause upgrade failures. A recent study [Crameri et al., 2007] identified broken dependencies and altered system-behavior as the *leading causes of upgrade failure*, followed by bugs in the new version and incompatibility with legacy configurations.

1.1.2 End-to-end upgrades in distributed systems

Prior work on software upgrades has been conducted independently, in separate research communities, and has focused on individual components of distributed systems. For instance, dynamic software updating (DSU) techniques [Segal and Frieder, 1993; Boyapati et al., 2003; Neamtiu et al., 2006] were developed by the programming-language community for modifying a program on-the-fly, without stopping it. DSU performs a single-host

online upgrade, where the entire system state is loaded in memory. In contrast, research on database-schema mapping and evolution [Ferrandina et al., 1995; Bernstein and Haas, 2008; Curino et al., 2008a] concentrates on migrating persistent data stored in a database and provides mechanisms commonly used for planning offline upgrades. The techniques proposed for upgrading distributed systems [Bloom, 1983; Kramer and Magee, 1990; Moser et al., 2000; Ajmani et al., 2006; Rellermeyer et al., 2008] focus on applications built on top of distributed-object middleware or component frameworks, where online upgrade is one of the mechanisms provided by the framework.

However, real-world distributed systems are not based on a single, homogeneous framework. Instead, they utilize three-tier architectures, with front-end servers that manage the client connections, middle-tier servers that implement the business logic and back-end servers that store the persistent data. An end-to-end upgrade replaces the old versions of the business logic and of the data schema with newer versions and requires coordinating the upgrade of multiple system components with the conversion of the persistent data objects. Industry best-practice recommendations [for example: Office of Government Commerce, 2007], advocate deploying the new version gradually, through *rolling upgrades* [Brewer, 2001; Microsoft Corporation, 2005; Oracle Corporation, 2008] that upgrade-and-reboot each host in the distributed system, one at a time, in a wave rolling through the data center.

During a rolling upgrade, the system's clients can interact with either the old version or the new version of the software. This requires the two versions to interact with each other in a compatible manner; for instance, if the upgraded components maintain persistent states, the old and new versions must undergo state synchronization during the upgrade. Current commercial tools that target rolling upgrades [for example: Microsoft Corporation, 2005; Oracle Corporation, 2008] provide no way for determining if the interactions between mixed versions are safe and leave these concerns to the application developers. 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 test and validate in advance [Segal, 2002]. Moreover, the failures that arise from such component-wise upgrades are not well understood, as the evaluations of previous upgrade mechanisms focus on performance and overhead rather than on the upgrade dependability.

1.2 The next step forward

This dissertation explores the following hypothesis:

> *The availability and reliability of distributed systems can be improved by performing software upgrades atomically, end-to-end, and by providing mechanisms for isolating faults that are specific to software upgrades.*

Specifically, a dependable online-upgrade mechanism should provide the *AIR properties*:

- ATOMICITY: At any time, the clients of the system-under-upgrade must access the full functionality of either the old version or the new version—but not both. The end-to-end upgrade must be an atomic operation.
- ISOLATION: The upgrade mechanism must not change, remove, or affect in any way the dependencies of the production system (including its performance, configuration settings and ability to access the data objects).
- RUNTIME-TESTING: The upgrade mechanism must allow testing the upgraded system under operational conditions.

I test this hypothesis by analyzing the necessity, the practicality and the generality of AIR upgrades. The AIR properties derive from empirical evidence of planned and unplanned downtime and from practical experience with system failures caused by the current upgrade mechanisms. I describe the design of Imago, a system that guarantees the AIR properties, and I discuss how the novel upgrade mechanisms incorporated in Imago can be applied to software upgrades in large-scale distributed systems. To assess whether Imago reaches its goal of improving the dependability of distributed systems, I develop an approach for benchmarking the availability and the reliability of systems that undergo online software upgrades. Finally, I investigate when these techniques are necessary by analyzing the consequences of relaxing the AIR properties.

1.3 AIR software upgrades

Improving the dependability of software upgrades requires connecting principles from several disciplines of computer science: distributed systems, databases, programming languages and software engineering. The main challenge in this endeavor is building a bridge

between the various experimental methods, techniques and practical considerations of these fields and distilling this knowledge into three abstract properties of dependable software upgrades. This dissertation traces the road to the AIR properties and describes the technical obstacles faced at each step.

The first hurdle is understanding the current causes of upgrade failure in distributed systems. While prior research, summarized in Chapter 2, suggests that most upgrade failures are due to problems in the operating procedure, rather than to software defects, the mechanisms of failure and their effects on the system are not sufficiently well understood to create a failure or simulation model for software upgrades [Liskov, 2001]. Chapter 3 presents an empirical study, which combines data from three independent sources, suggesting that upgrades fail primarily because of unavoidable human errors in the upgrade procedure. These errors break hidden dependencies in the system under upgrade, *e.g.*, by incorrectly specifying the location of certain services, by creating database-schema mismatches, or by introducing conflicts among shared libraries. Through statistical cluster analysis, I also identify what these failures have in common and I propose a novel, upgrade-centric, fault model.

The lack of insight into upgrade failures raises a subtler challenge. We currently do not know how to evaluate the dependability of distributed systems that undergo software upgrades. The existing information on upgrade failures is largely anecdotal, and real-world data is difficult to obtain due to the sensitivity of the topic. The known examples of failed upgrades (*e.g.* the ones described in Section 1.1) cannot be replicated because there is not enough data on the system configurations, network topologies, errors encountered, *etc.* As a result, the evaluations of previous upgrade mechanisms focus on their performance and overhead, rather than on the upgrade dependability. This challenge is addressed in Chapter 7, which introduces a dependability-benchmarking approach for upgrade mechanisms. The benchmark is based on fault-injection experiments driven by the upgrade-centric fault model from Chapter 3.

However, even successful upgrades often require planned downtime for changing the data schema or for migrating to a different data store. Because some conversions are difficult to perform on the fly, in the face of live workloads, and owing to concerns about overloading the production system, complex data conversions currently impose downtime on upgrade. Chapter 4 investigates the leading causes of such planned downtime.

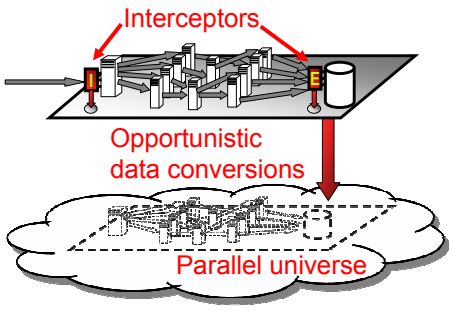


Figure 1.2. Conceptual overview of AIR upgrades.

Understanding the leading causes of planned and unplanned downtime leads to the formalization of the AIR properties, in Chapter 5. These properties are necessary for compensating shortcomings in the existing mechanisms for software upgrade. However, enforcing the AIR properties requires a new approach for performing software upgrades. Chapter 6 describes the practical trade-offs involved in the implementation of an AIR-compliant system, called Imago, and the mechanisms that allow extending this system to provide upgrades-as-a-service for a wide class of applications. Chapter 7 shows that, through the AIR properties, Imago is able to reduce both the planned and the unplanned downtime due to software upgrades.

The final challenge is determining whether the AIR properties are not only sufficient, but necessary as well. The lessons learned from the design and implementation of Imago suggest that the AIR properties might be too strong or too expensive to provide in some cases. Chapters 8 and 9 analyze the consequences of relaxing the ISOLATION and ATOMICITY properties, respectively, during upgrades of large-scale distributed systems.

Figure 1.2 presents a conceptual overview of AIR upgrades. The key idea is to isolate the production system from the upgrade operations in order to avoid breaking hidden dependencies. This is achieved by installing the new version in a *parallel universe*, which uses resources leased, temporarily, from a cloud-computing infrastructure. The persistent data is transferred to the new version, and the necessary schema and data conversions are performed *opportunistically*. Instead of modifying the components of the system-under-upgrade in place, along with all their dependencies, a few key points are instrumented to monitor the live workload: the *ingress points*, where the clients direct their requests (*e.g.* the Web proxies in the front-end), and the *egress points*, where the system stores its persistent data (*e.g.* the database in the back end). This approach targets distributed-system

upgrades that implement major changes and that require long-running data migrations. The end-to-end upgrade is an atomic operation, executed online even when performing complex schema and data conversions. AIR upgrades also enable testing system states that emerge only at runtime, under operational conditions, and that cannot be adequately examined offline, during development or the other stages of the software life cycle.

1.4 Contributions

This dissertation explores the opportunities for improving the dependability of distributed systems by focusing on dependability's weakest link: software upgrades. An important goal of this research is the development of basic principles and of reusable upgrade mechanisms that tolerate the most common upgrade-related faults. I present here several key findings that I believe are applicable, in practice, to the design and maintenance of various distributed systems and that should guide future research on software upgrades in this setting.

Following an incorrect upgrade procedure accounts for less than half of the upgrade faults recorded. The most frequent procedure violations are omissions, where upgrade administrators skip a required step in the upgrade procedure and which account for 22% of all procedural errors. In 56% of cases, however, the administrators introduce upgrade faults despite correctly following the mandated procedure. Procedure violations alone cannot explain the occurrences of failed upgrades, which suggests that software upgrades do not fail, predominantly, because of avoidable human errors.

Breaking hidden dependencies is the leading cause of upgrade failure. Distributed systems often include hidden dependencies, *e.g.* service locations specified incorrectly, shared-library conflicts, database-schema mismatches between the middle tier and the back-end. These dependencies remain hidden because they cannot be detected automatically or they are overlooked owing to their complexity. 85% of all the upgrade faults recorded break hidden dependencies. Procedure violations often occur because upgrade administrators are not aware of certain hidden dependencies.

There are four types of faults the commonly occur during software upgrades: (1) simple configuration errors (*e.g.* typos); (2) semantic configuration errors (*e.g.* misunderstood effects of parameters); (3) broken environmental dependencies (*e.g.* library or port conflicts); and (4) data-access errors, which render the persistent data partially unavailable. Faults of types 1 and 4 occur in up to 45% of software upgrades. This classification defines an upgrade-centric fault model.

Changes in data formats or schemata represent the leading cause of planned downtime. When an upgrade implements major changes to the data schema, the old version of the application logic, in the middle tier, can no longer query the new version of the schema, in the back-end. In consequence, the data store and the application logic must be upgraded together, atomically. This prevents upgrade mechanisms that gradually replace the old version, such as rolling upgrades. Because the schema conversion is a long-running process, the upgrade requires planned downtime that typically ranges from tens of hours to several days.

The AIR properties address the leading causes of both planned and unplanned downtime. The Isolation property prevents failures that result from breaking hidden dependencies during the upgrade, because the upgrade operations access the production system in a non-intrusive, read-only manner. The Atomicity and Runtime-testing properties imply that the system must not include mixed, interacting versions. The lack of mixed versions enables long-running data conversions during an online upgrade; a degraded functionality is necessary only during the atomic switchover to the new version.

The dependability of software upgrades can be improved by trading resource overhead. Imago requires additional hardware and storage resources for building a parallel universe, in order to perform the upgrade. This allows Imago to eliminate the internal single-points-of-failure for upgrade faults and to avoid creating states with mixed, interacting versions. This trade-off is based on the observation that the potential cost of downtime in modern distributed systems

offsets the costs of new hardware or of leasing resources from a public cloud-computing infrastructure.

Performing upgrades in a manner that is agnostic to dependencies and to mixed-version interactions enables the creation of generic upgrade mechanisms. This approach introduces a separation of concerns between the functional aspects of the upgrade—*e.g.* data conversions, which require domain-specific code—and the mechanisms for implementing an online upgrade—*e.g.* performing an atomic switchover. Most of Imago's code is reusable for other upgrade scenarios. This enables an upgrade-as-a-service approach, where most of the hardware and software resources needed for a distributed-system upgrade are provided by a third party.

It is possible to benchmark the dependability of software-upgrade mechanisms. Unlike for performance benchmarks, generating representative input workloads is not sufficient because software upgrades alter the behavior of the systems being evaluated. A benchmark focusing on dependability should specify upgrade workflows that commonly cause unavailability and failures during software upgrades in real-world systems. Understanding the leading causes of upgrade failures and the leading causes of planned downtime allows conducting fault-injection experiments that produce representative results for the dependability of the upgrade mechanisms evaluated.

Upgrades in a Service-Oriented Architecture (SOA), provide a relaxed version of the Isolation **property.** SOA is widely used for isolating services in real-world distributed systems, but it does not eliminate all the possible hidden dependencies. For instance, service-level objectives managed by independent, third-party managers cannot always be reconciled, and the lack of coordination among managers can cause oscillatory instability. Guaranteeing Isolation in SOA requires a distributed framework for impact assessment, where the impact of software upgrades and other planned changes is estimated by the domain-specific managers, while a centralized component seeks the most opportune time to execute these operations.

The lack of upgrade ATOMICITY can lead to race conditions involving multiple versions of the software. Such mixed-version races can occur, during rolling upgrades, in systems that communicate across administrative domains using asynchronous messaging. While this race condition has not been characterized before, two real-world examples of upgrade failure can be traced back to mixed-version races. In the near future, many distributed systems that rely on cloud-based resources and span multiple administrative domains are likely to be affected by mixed-version races. However, the likelihood of occurrence and the expected impact of mixed-version races can be easily estimated through an analytical risk model, which allows system administrators to decide whether to upgrade or not to upgrade.

This dissertation includes both empirical studies of upgrades in existing systems and first-hand observations from the design and evaluation of a novel system for dependable, online upgrades. The main contributions are two-fold: (i) I identify the leading causes of upgrade failure—breaking hidden dependencies—and of planned downtime—changing database schemas—in distributed enterprise systems; and (ii) I describe the design principles of dependable upgrade mechanisms. The AIR properties provide a framework for reasoning about the impact of online software upgrades: ATOMICITY implies that the system-under-upgrade does not enter states with mixed versions that synchronize their states on-the-fly, which reduces the need for planned downtime, and ISOLATION prevents breaking hidden dependencies, which reduces the upgrade failures that lead to unplanned downtime. I show that relaxing these properties can introduce unexpected runtime behaviors, such as oscillatory instability or mixed-version races. However, in some cases, technical solutions to these unexpected behaviors can be developed, in order to avoid the overhead of strong AIR properties and to make dependable upgrades more practical. Additionally, RUNTIME-TESTING will become important in the near future for testing system states that emerge in the deployment environment and that may be unpredictable at design-time.

The mechanisms for performing software upgrades and the method for benchmarking such mechanisms described in this dissertation also represent useful, although secondary, contributions.

1.4.1 Limitations of scope

This dissertation does not focus on disseminating fine-grained updates, such as bug fixes or security patches. Instead, it targets major system upgrades, which involve behavioral changes and persistent-data conversions. While the goal of this dissertation is to improve the dependability of distributed systems that undergo such complex software upgrades, I do not claim to eliminate all the possible sources of downtime or upgrade failures; in particular, I do not focus on upgrade failures that result from software defects that are introduced either during the development or the requirement-elicitation phases of the software life cycle. This dissertation analyzes faults that occur when a complex software system is modified in its deployment environment. To explore the various trade-offs required for enforcing the AIR properties in a real deployment, I do not concentrate on performing upgrades in place, without the need for additional resources, or on implementing upgrades that are fully transparent to the clients. Moreover, while the AIR properties are generic and accommodate implementations targeting other kinds of distributed systems, the upgrade mechanisms and the evaluations described in this dissertation focus on three-tier enterprise architectures.

> It is possible to replace an object with another object, without stopping the system and without requiring great programming skills from the developer.
>
> P. Narasimhan, Ph.D. Dissertation, 1999, §9.1 Outstanding challenges

Chapter 2

Related Work

Actively used software must be modified continuously to ensure its utility and safety. Fixing bugs, adding new features, removing obsolete features—all involve upgrading existing software. Sometimes the goal of the upgrade is to migrate to a new platform (*e.g.*, a cloud-computing infrastructure), because the previous platform has reached its end-of-life or for efficiency reasons. In enterprise systems, business reasons sometimes mandate switching vendors, while responding to customer expectations and conforming with government regulations can require new functionality.

Some of these software upgrades focus on individual components, such as the application code, the middleware framework or the persistent-data formats. In distributed systems, upgrades typically require coordinating the changes applied to several components. Because distributed systems typically include hardware redundancy and fault-tolerance mechanisms, which allow individual components to be temporarily unavailable (while they are being upgraded), previous research has dedicated considerable attention to performing online upgrades in single-host systems. However, distributed systems include more complex dependencies and exhibit different failure modes during and after software upgrades. In this chapter, I review the research and the common practices that are related to the contributions of this dissertation.

2.1 Causes of upgrade-induced downtime

While some upgrades fail because the new version includes software defects, Crameri et al. [2007] present the results of a survey suggesting that *broken dependencies* and *altered system-behavior* as the leading causes of upgrade failures and unplanned downtime. Similarly,

Curino et al. [2008b] present the largest study of database schema evolution in information systems, and they suggest that many of these *schema changes* impose planned downtime on upgrade.

Anderson [2000] describes three mechanisms that are commonly responsible for breaking dependencies on dynamically-linked libraries (DLLs) in the Windows NT operating system—a phenomenon colloquially known as "DLL Hell". More recently, Dig and Johnson [2006] examine how the evolution of APIs can impact upgrades by breaking dependencies on third-party Java frameworks and libraries. They conclude that between 81%–100% of the breaking API changes are *code refactorings* (reorganizations of the program structure) and that less than 30% are intended behavioral changes (modified application semantics).

API compatibility is not the only class of dependencies that can be broken during an upgrade. Upgrades in distributed systems usually require a complex process that involves hardware and software additions, reconfigurations, and data migrations. Oppenheimer et al. [2003] study 100+ *post-mortem* reports of user visible failures from three Internet services. They classify failures by location (front-end, back-end and network) and by the root cause of the failure (operator error, software fault, hardware fault). Most failures reported occurred during change-management tasks, such as scaling or replacing nodes and deploying or upgrading software. Nagaraja et al. [2004] report the results of a user study with 21 system administrators, who are asked to perform various change-management tasks, and observe seven classes of faults: global misconfiguration, local misconfiguration, start of wrong software version, unnecessary restart of software component, incorrect restart, unnecessary hardware replacement, wrong choice of hardware component. Oliveira et al. [2006] 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. While the database administrators who responded to the survey spend only 46% of their time performing change-management tasks, all the faults reported can occur during an upgrade. Keller et al. [2008] study configuration errors and classify them according to their relationship with the format of the configuration file (typographical, structural or semantic) and to the cognitive level where they occur (skill, rule or knowledge).

These models do not constitute a rigorous taxonomy of upgrade faults. Some classifications are too coarse-grained (*e.g.*, the fault location [Oppenheimer et al., 2003]) and do not

provide sufficient information about the fault. Other classifications (*e.g.*, the breaking API changes [Dig and Johnson, 2006] and the typographical/structural/semantic configuration errors [Keller et al., 2008]) are relevant for only a subset of the upgrade faults (broken dependencies on third-party frameworks and configuration-file modifications, respectively). In many cases, the fault categories are not disjoint and the criteria for establishing these categories are not clearly stated. Moreover, the previous studies of database schema evolution [for example: Curino et al., 2008b] do not identify which schema changes require planned downtime because they are difficult to integrate in an online upgrade.

2.2 Properties of software upgrades

Fabry [1976] identifies the possibility of *online upgades*, which are needed in systems with high-availability requirements, such as the early computer programs for processing airline reservations. This need is increasingly important for the computer systems that we build today. Fabry also remarks that, aside from ensuring that upgrades can be executed online, we must also provide guarantees that the program will operated correctly after the upgrade is deployed.

Kramer and Magee [1990] argue that software upgrades should be specified as structural changes, such as component connections and disconnections, and that this declarative, high-level specification should be separate from functional concerns, such as transferring the state to the new version. This separation of concerns allows the authors to define general rules for software upgrades, such as the *minimal control API* that a component should provide to enable upgrades: `passivate`, `assert(active/passive)`, `activate`, `link`, `unlink`. Moreover, each component should be prepared to handle state transfers if necessary.

Boyapati et al. [2003] introduce a *modularity* property for software upgrades, which enables reasoning locally about the correctness of online upgrades, assuming only the interfaces and invariants of the old version. This is similar to the way programmers reason about modular code in object-oriented programming languages. While an upgrading mechanism might delay the execution of upgrades, to avoid downtime, modularity is guaranteed if the upgrades appear to run in the order in which they are deployed, with respect to other application transactions and to subsequent upgrades. Neamtiu et al. [2006] further

show that, when updating programs written in procedural languages that are not object-oriented, such as C, the upgrade mechanism must ensure that old code does not access new data, a property called *representation consistency*. Focusing on distributed-system upgrades, Ajmani et al. [2006] define a correctness requirement for systems operating with *mixed versions*: each invocation of a version must reflect the effects of all earlier events processed by the other versions, in the order in which they occurred. Effectively, this requirement implies that the states of all versions are synchronized and that some invocations must be disallowed temporarily, where invariants tying two versions together cannot be established.

A common theme of programming-language research on dynamic software updates is identifying the points in the program's execution where the update can safely take place. For example, *activeness safety* ensures that active code, which remains on the program stack—*e.g.*, long-running loops or the `main()` function—or which might invoke other active functions, must not be updated [Segal and Frieder, 1993]. Stoyle et al. [2007] show that dynamic updates can change the program's structure and remain type-safe by guaranteeing *con-freeness safety*: an update can be applied to a type if none of the type's instances are in the continuation of the current update point. However, Gupta [1994] proves that, in general, finding such *safe update points* is undecidable.

Database research on schema evolution focuses on the transformations applied to persistent data, rather than on the implications of online upgrades. For example, Fagin [2007] identifies the schema mappings that are *uniquely invertible*, allowing the effects of an upgrade to be reversed. Similarly, Curino et al. [2008a] demonstrate a system that checks automatically which mappings are *information preserving* and which would result in a loss of data.

2.3 Approaches for dependable upgrade

To avoid degrading the system availability during software upgrades, mechanisms for online-upgrade[1] have been introduced, targeting both single-host and distributed systems. Various techniques aim to improve the reliability of software upgrades, ranging from dependency-tracking to sandbox-testing.

[1] In some references, online upgrades are referred to as *zero-downtime upgrades* or *live upgrades*.

2.3.1 Dependability through dependency tracking

To prevent breaking dependencies during an upgrade, modern operating systems maintain repositories of packaged software components. In addition to the software, a package contains executable configuration scripts and metadata describing the dependency and conflict relationships with other packages. For example, in Debian Linux there are approximately 25,000 packages, 50,000 scripts and 85,000 inter-package dependencies [LaBelle and Wallingford, 2006; Di Cosmo et al., 2008], which define intricate dependency graphs (see Figure 1.1 for a sample subset of such a graph). To ensure that the system retains a correct configuration after the upgrade, the package-management tools traverse the dependency graph to determine which new packages have to be installed, in order to satisfy all the dependencies, and which existing packages must be removed, in order to avoid any conflicts.

The Windows Update Agent [Microsoft Developer Network, 2001], for instance, determines the updates that are already installed by querying a local data store and the host's configuration registry. The agent then sends this information to the Windows Update server, which resolves the dependencies and determines which updates can be installed on the client host [Leyden, 2003]. In the Linux world, RedHat's YUM [Brown and Pickard, 2003], Debian's APT [Silva, 2005] and Gentoo's Portage [Vermeulen et al., 2007] download the list of available packages from the remote repository perform dependency resolution on the local host. After determining which packages to install, Portage downloads the source code and compiles it locally, while YUM and APT download the binary software components. Appupdater [McNab and Bryan, 2009] extends the scope of this approach beyond updating core components of operating systems by providing a general purpose mechanism for automatically updating Windows-based application software.

Some package-management systems (*e.g.* Windows Update, Portage) try to minimize the risk of conflicts in the deployment environment by allowing multiple versions a shared library to operate side-by-side in the deployment environment. To further improve the dependability of software upgrades, Dolstra and Löh [2008] propose Nix, a purely functional package-management system where software installations do not have side-effects and system configurations never change after they have been built. Nix packages are built from mathematical expressions, which capture the package's complete dependencies, and they are installed in a local data store, which can accommodate multiple versions of the same

package. Nix upgrades are guaranteed to be atomic because they are committed through a symbolic-link replacement (an atomic operation in Unix). Di Cosmo et al. [2008], however, point out that installation side-effects cannot be eliminated in practice because the configuration scripts modify the system state and execute operations that cannot be rolled back easily.

Furthermore, resolving dependencies is an NP-complete problem (see Appendix A). Existing package management tools use heuristic algorithms [Silva, 2005; Brown and Pickard, 2003] to ensure that the search for a correct configuration terminates in a timely fashion, or they rely on SAT solvers [Tucker et al., 2007; Di Cosmo, 2005] to guarantee that a solution will be found for all installable packages. Heuristics-based approaches might fail to find a solution where one exists, while, for certain corner cases, the run time of SAT solvers is exponential in the size of the repository [Tucker et al., 2007]. These approaches seem adequate for the current sizes and structures of software repositories: APT fails to find a solution for 0.61% of installable packages while the approaches based on SAT solvers run for less than a minute [Mancinelli et al., 2006; Tucker et al., 2007]). However, as the amount of dependency metadata increases, these techniques for resolving dependencies will soon reach the limits imposed by NP-completeness. For instance, taking into account the dependencies on configuration settings would increase the amount of metadata by an order of magnitude. Distributed systems include additional dependencies on APIs, persistent data objects, communication protocols, Internet routes or performance levels.

Ultimately, package-management tools perform software upgrades dependably only when they have the complete dependency information about the system they are managing. However, package repositories are known to include references to nonexistent packages (see Figure 1.1), libraries that declare to have the same version but differ at binary level [Hart and D'Amelia, 2002], or packages that cannot be installed due to unresolvable dependency-and-conflict cycles [Mancinelli et al., 2006]. In practice, dependencies cannot always be detected automatically. For example, many applications rely on the APIs exported by third-party libraries. Dig and Johnson [2006] find that over 80% of breaking changes introduced during the API evolution of Java libraries are due to refactorigs (modifications of the program structure) and that some refactorings could be detected automatically, through a combination of syntactic and semantic analyses [Dig et al., 2006]. However, this approach does not handle the API evolution resulting from intended behavioral

changes, which modify the semantics of the application. In Windows, some libraries have embedded version information, and other programs store this information in the host's configuration registry. Appupdater maintains a local database with the hash values of installed software and maps hash values to particular versions of all the applications it manages [McNab and Bryan, 2009]. Galapagos [Magoutis et al., 2008] parses configuration files to discover the relationships among storage objects from different tiers and abstraction layers of a distributed system. These approaches perform a static analysis of the system do not guarantee the discovery of all the dependencies. Similarly, runtime monitoring [Dunagan et al., 2004] or active perturbation [Brown et al., 2001] cannot provide a complete coverage of all the factors that might influence the behavior of a distributed system. Approaches for tracking dependencies rely on the correctness of metadata that is partially maintained, manually, by teams of developers and quality-assurance engineers through a time-intensive and error-prone process.

Dependency-tracking approaches aim to prevent system failures that result from broken dependencies. However, dependable software upgrades must also avoid planned downtime. This can be achieved by performing an upgrade online, without powering off the system.

2.3.2 Online upgrade in single-host systems

Dynamic software updates (DSU) aim to modify a running program on-the-fly, without affecting the system's availability. DSU techniques typically focus on updating systems spanning a single host, where the entire state of the system is loaded in the host's memory. This approach has been studied extensively during the past 35 years [from Fabry, 1976 to Bhattacharya and Neamtiu, 2010] and has been applied successfully in practice to upgrade special-purpose components (*e.g.* the modules of a telephone switch [Toy, 1992]), system libraries [Buban et al., 2004], OS kernel modules [Baumann et al., 2007] and even the kernel itself [Arnold and Kaashoek, 2009]. Typically, these approaches support changes to the implementations, but not the signatures of the functions that they update. Perhaps the most advanced DSU techniques are implemented in the Ginseng system, of Neamtiu et al. [2006], which uses static analysis to ensure the safety and timeliness of updates (*e.g.*, establishing constraints to prevent old code from accessing new data) and supports all the changes required for updating server programs (three years' worth of releases for the Very

Secure FTP daemon, the OpenSSH `sshd` daemon and the GNU Zebra routing software), without dropping the client connections.

The PODUS system [Segal and Frieder, 1989a, 1993] introduced generic DSU techniques for applying updates at the *procedure granularity—e.g.*, replacing C functions in a running program, even when the function prototype has changed in the new version—and identified two fundamental challenges for DSU: performing state transformations and updating active code. In object-oriented languages, some updates can be performed at the class level, without converting the object instances from the heap memory. For example, the Java-like UpgradeJ language [Bierman et al., 2008] supports three forms of updates: (i) adding new classes, (ii) changing the implementation of existing classes, and (iii) adding new fields and extending the signatures of existing methods with additional parameters. Tempero et al. [2008] show that these mechanisms would enable performing between 10% and 65% of the changes occurring in real-world Java applications. Most DSU techniques employ *transfer functions* [Boyapati et al., 2003; Neamtiu et al., 2006] to convert the program state into the format required by the new version. UpStare [Makris and Bazzi, 2009] introduces techniques for *stack reconstruction*, which allow changing active code by manipulating the program stack at the time of the update. In general, DSU techniques require additional development effort, *e.g.*, source-code annotations for specifying update points or transfer-function implementations. Some of these challenges can be avoided when updating programs written in a functional programming language, such as Erlang [Armstrong et al., 1996].

Because DSU can introduce new sources of software defects (*e.g.*, if the update is applied at the wrong time), Hayden et al. [2009] propose verifying the upgrade through *testing*. This approach instruments the application to trace possible update points, groups them into equivalence classes and tries to apply the update at a point in each class to determine whether a conflict would be created. Conflicts are detected if the program-trace changes when an update is applied. Reflecting on the lessons learned from the PODUS system, Segal [2002] remarks that it is difficult to reason about the behavior of systems undergoing dynamic updates. During the update, the system behavior is not guaranteed to conform to the specification of either the old or the new version of the software, which prevents the adoption of DSU in systems with strict certification requirements.

Unlike DSU, approaches based on virtualization aim to isolate the new version from the old one, in order to avoid breaking dependencies. These approaches focus on upgrading

operating systems, and they provide mechanisms for encapsulating the state of running applications and for migrating them to a different virtual machine during the upgrade. For example, Lowell et al. [2004] propose upgrading operating systems using lightweight virtual machines. They describe the Microvisor virtual machine monitor, which allows a full, *devirtualized* access to the physical hardware during normal operation. During a software upgrade, the running applications are migrated to a separate virtual machine. To reduce the overhead, Microvisor virtualizes only the CPU and relies on additional hardware peripherals (*e.g.* I/O devices) for launching the upgrading virtual machine. The applications must either be stateless or they must checkpointing mechanism for saving their state and restoring it in the virtual machine. To facilitate this migration process, Potter and Nieh [2005] propose AutoPod, which virtualizes the OS's system calls, allowing applications to move among *location-independent pods*. This functionality is used for upgrading the OS without stopping the running applications. However, AutoPod assumes that the new version of the OS doesn't introduce breaking API changes for the system calls, which limits the upgrades to minor versions of the kernel (*e.g.*, the Linux 2.4 series) that are not allowed to break application compatibility.

2.3.3 Online upgrades in distributed systems

The earliest work on distributed-system upgrades [Bloom, 1983] relies on the *crash recovery* and *state transfer* mechanisms from the Argus system [Liskov, 1988], which were originally developed for coping with crash faults and network partitions. Similarly, the Eternal system [Moser et al., 1998; Narasimhan, 1999], which provides fault tolerance to legacy CORBA applications by redirecting the message exchanges to a group-communication protocol, can leverage this mechanism to coordinate the distributed upgrade [Tewksbury et al., 2001]. Eternal creates *intermediate versions*, supporting both the old and the new interfaces of the object being upgraded. An intermediate version acts as a staging area, where the old and the new versions can coexist and where state can be transferred between them, in the presence of a live workload. If X corresponds to the old version, Y to the new version and XY to the intermediate version, the upgrade process consists of four steps: (i) migrate each replica of X to a replica of XY, one a time; (ii) perform state transfer within each replica of XY, in the presence of live requests; (iii) migrate each replica of XY to a replica of Y, one a time; and (iv) orchestrate a system-wide *atomic switchover* that causes replicas of X and

replicas of XY to become defunct. Only replicas of Y still exist after the switchover point. Tewksbury et al. 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 [Kramer and Magee, 1985] upgrades component-based systems through architectural reconfigurations, *i.e.* changing components and connectors. When upgrading a component, Conic enforces quiescence by invoking a control API [Kramer and Magee, 1990; see also Section 2.2] that allows *passivating* all of the component's inbound nodes. Conic automatically determines the correct sequence of control-API invocations required when upgrading several components. These principles are reflected in modern component frameworks such as R-OSGi [Rellermeyer et al., 2007], which upgrade a component along with the *transitive closure of its inbound dependencies* [Rellermeyer et al., 2008].

In the absence of fault-tolerance mechanisms or control APIs, the PODUS system [Segal and Frieder, 1989a, 1993] establishes simple rules for coordinating a distributed-system upgrade, such as upgrading servers before their clients [Segal and Frieder, 1989b]. 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 [Ajmani et al., 2006] 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 upgrade. Bhattacharya and Neamtiu [2010] propose avoiding mixed-version states by keeping track of the safe update points on the hosts of the distributed system—including the application servers and the database—and executing an atomic switchover when all the hosts are ready to deploy the upgrade simultaneously.

Crameri et al. [2007] suggest that the risk of upgrade failure can be reduced by *testing* new or updated packages in a wide variety of user environments and by *staging the deployment* of upgrades to increasingly dissimilar environments. This approach, implemented in a system called Mirage, accelerates the deployment by clustering similar user environments and uses automated tools to discover the characteristics of each environment: each file is represented by a fingerprint, which is determined using semantic parsers that capture the significant information (*e.g.*, configuration settings, versions of shared libraries). Mirage

requires accurate semantic parsers for each file type in order to produce meaningful clusterings. Tucek et al. [2009] propose to expedite the testing of multiple software versions through a technique called *delta execution*. This technique merges the redundant executions of nearly identical versions, splitting the execution when the control flow reaches different code segments or processes different data.

These testing approaches focus on finding software defects. To prevent upgrade failures due to misconfigurations or operator errors, other testing approaches focus on executing multiple versions side-by-side, in a sandboxed environment, and on comparing their outputs. Nagaraja et al. [2004] propose a technique for detecting operator errors by performing upgrades or configuration changes in a *validation slice*, isolated from the production system. The upgraded components are tested using the live workload or pre-recorded traces. This approach requires component specific *inbound- and outbound-proxies* for recording and replaying the requests and replies received by each component-under-upgrade. If changes span more than one node, multiple components (excluding the database) can be validated at the same time. Oliveira et al. [2006] extend this approach by performing change operations on an up-to-date replica of the production database. The tests employed in these approaches compare the performance (*e.g.* latency and throughput) of the two versions and compute the edit distance between their outputs. Because they operate at component granularity, these approaches require a detailed knowledge of the system's architecture and queuing paths. For example, implementing the inbound/outbound proxies requires understanding the, often proprietary, protocols employed by the front-end servers and by the database to communicate with the application servers in the middle tier. Splitter [Ding et al., 2010] is an approach for validating the behavior of enterprise systems after their migration to a virtualized infrastructure, in order to ensure that the system continues to function correctly in the new environment. While Splitter does not focus on upgrading the system software, it introduces *ranking heuristics* and statistical techniques for comparing the outputs of identical software versions running in two different infrastructures—one physical and one virtualized. Splitter's proxies rely only on the semantics of HTTP requests and replies (*e.g.*, for mapping cookies, URL parameters and AJAX callback arguments between the two systems). Similarly, Tan et al. [2005] describe a "server Tee" that duplicates the client requests in order to compare the outputs of a file server using the stateless NSFv3 protocol against a reference implementation. A common challenge in these approaches is

to account for the non-deterministic behavior of the systems being compared, such as routing requests to different middle-tier servers or executing concurrent requests in different orders. To reduce the rate of false-positive warnings from the testing harness, the existing approaches prevent transaction concurrency by enforcing a common serialization of database queries for both systems [Oliveira et al., 2006; Ding et al., 2010] or do not take into account the the non-comparable responses of concurrent writes to the same data block [Tan et al., 2005].

Other validation techniques have been proposed for reducing the risk of online upgrades in spite of unknown software defects in the new version. The Simplex architecture [Sha et al., 1996] for upgrading real-time control systems executes the new version in parallel with a simple, and known to be correct, control algorithm, which cross-checks its outputs for validating the upgrade. The two versions need not produce identical results, but they must satisfy a *behavioral model*. For upgrading spacecraft software during long deep-space missions, the Guarded Software Upgrades framework [Tai et al., 2002] assesses the *confidence in the correctness* of each upgraded component. The components communicate exclusively through message-passing and the framework uses distributed checkpointing and rollback algorithms to prevent, or recover from, state contamination due to messages exchanged with low-confidence components. More recently, a similar idea was proposed for generating fault-tolerant compositions of Web Services that undergo online upgrades, by invoking multiple versions of a service in parallel and by using the confidence in correctness to reason about future failure rates [Gorbenko et al., 2005].

2.3.4 Industry best-practices for software upgrades

Industry best-practices, such as the Information Technology Infrastructure Library (ITIL) [Office of Government Commerce, 2007], recommend a phased deployment of software upgrades. This is usually realized through *rolling upgrades*, which upgrade-and-then-reboot each host in a wave rolling through the distributed system [Brewer, 2001]. 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 [Downing, 2008]. In such cases, the enterprise application will include a mix of ver-

sions that operate concurrently at different installation sites, in order to avoid placing any site in a read-only mode or introducing state divergence.

These requirements have motivated the introduction of several commercial products performing rolling upgrades [Microsoft Corporation, 2005; Oracle Corporation, 2008] and for synchronizing the persistent state of two versions [Choi, 2009]. These commercial products provide no way for determining if the interactions between mixed versions are safe and leave these concerns to the application developers.

2.4 Dependability benchmarking for software upgrades

Existing evaluations of software 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 [Beattie et al., 2002; Oppenheimer et al., 2003], surveys [Oliveira et al., 2006; Crameri et al., 2007], fault injection [Nagaraja et al., 2004; Oliveira et al., 2006] and direct experimentation [Crameri et al., 2007; Zheng et al., 2009], have been proposed for assessing the effectiveness of previous upgrade mechanisms in reducing the number of upgrade failures.

Beattie et al. [2002] 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. [2003] 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. Nagaraja et al. [2004] propose injecting operator mistakes observed during their user study of 21 system administrators. In contrast, Oliveira et al. [2006] propose injecting synthetically-generated mistakes, designed based on the results of a survey of 51 database administrators. The survey also suggests that 84% of schema and data conversions are tested and deployed in different environments, which increases the risk of upgrade failure. Crameri et al. [2007] 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. The authors also propose reducing the risk of failures by testing the upgrades in their deployment environments. Zheng et al. [2009] propose running experiments with

different configurations, in a virtualized data center, in order to reduce the cost of answering "what-if" questions about the configuration changes.

2.5 Impact assessment for online upgrades

Autonomic computing [Kephart and Chess, 2003; IBM Corporation, 2006] attempts to reduce the impact of operator mistakes by automatic many system-management tasks in distributed enterprise systems. This approach relies on autonomic managers that incorporate the domain knowledge to answer "what-if" questions about the impact of various change operations, such as software upgrades. For example, CHAMPS [Keller et al., 2004] features a complex dependency-tracking framework and focuses on minimizing the planned downtime required for completing a complex upgrade. Keller et al. formulate this problem as the optimization of a generic cost function given a set of constraints, which represent the upgrade impact (*e.g.* system unavailability). CHAMPS provides a centralized approach for both scheduling and impact analysis, and it relies on detailed, explicit representations of the complex dependencies between the components of the distributed system.

Decentralized mechanisms for impact assessment have also been proposed, in the context of autonomic management of storage systems. Hippodrome [Anderson et al., 2002] refines the initial configuration of a storage system through an iterative process, using a *model of the system's performance* model to estimate the throughput and capacity of a particular configuration. The K2 middleware [Golding and Wong, 2006] goes further in distributing the autonomic management functionality by eliminating the centralized decision- maker and allowing individual *allocation pools* to manage their own objectives. In K2, distributed decision algorithms determine the goal configuration and the allocation pools start moving in that direction; if conditions change part-way through reconfiguration, the system changes its direction without having to invalidate the previous plan.

Thereska et al. [2006] describe a *resource advisor* for predicting the impact of data placement and encoding choices on performance. The advisor has a hierarchical design, based on several "what-if" modules (*e.g.*, for predicting the CPU, network and disk delays and cache hit rates) that can be combined together for end-to-end latency and throughput predictions. The resource advisor continuously monitors the infrastructure and uses historical

data to over-provision the system based on the peak loads observed. The authors report fewer than 15% prediction errors, in most cases.

Quand une regle est fort composée, ce qui luy est conforme passe pour irrégulier.

G. W. Leibniz, *Discours de métaphysique*, 1686.

Chapter 3

Why Do Software Upgrades Fail?

WHILE fault-tolerance mechanisms focus almost entirely on responding to, avoiding, or tolerating unexpected faults or security violations, system unavailability is usually the result of planned events, such as upgrades. A 2007 survey of 50 system administrators from multiple countries (82% of whom had more than five years of experience) concluded that, on average, 8.6% of upgrades fail, with some administrators reporting failure rates up to 50% [Crameri et al., 2007]. The survey identified *broken dependencies and altered system-behavior* as the leading causes of upgrade failure, followed by bugs in the new version and incompatibility with legacy configurations. This suggests that most upgrade failures are not due to software defects, but to *faults that affect the upgrade procedure*.

A system *failure* prevents the system from providing correct service, while a *fault* affects a single component and might be masked or tolerated by the distributed system. The current severity of broken-dependency faults may come as a surprise because dependencies on third-party components have been studied extensively. For example, it has long been known that dependencies among program modules can cause a ripple effect during software maintenance, because a few minor-bug fixes require an exponential number of changes in other parts of the code [Yau and Collofello, 1980]. Recent advances in understanding the impact of software defects [Sullivan and Chillarege, 1991; Li et al., 2006] and their relation to source-code changes [Dig and Johnson, 2006; Nagappan et al., 2010] have allowed us to create automated tools for software development and to build software systems that approach 1 billion lines of code [Northrop et al., 2006].

Development-time techniques, as well as tools for package management, focus exclusively on functional dependencies among software components. However, the behavior of

a distributed system is also influenced by non-functional and runtime dependencies on performance levels, timing properties, data objects, communication protocols, Internet routes or configuration settings, which cannot always be detected or handled automatically (see Section 1.1.1). In consequence, sometimes *dependencies in the deployment environment remain hidden* from the system administrators and can be broken during a software upgrade.

This chapter presents an empirical study showing that current approaches for upgrading distributed systems, which rely on tracking dependencies for preserving system integrity before and after the upgrade, are vulnerable to common faults that occur during software upgrades. These upgrade-specific faults persist in modern distributed systems because dependency tracking is often incomplete and upgrades are not atomic operations, which introduces the risk of breaking hidden dependencies.

I propose a novel upgrade-centric fault model, based on data from three independent sources: a field study of the Apache bug reports, a user study of system administrators engaged in change-management tasks, and a survey of database administrators (DBA). This fault model focuses on the impact of procedural errors rather than software defects. There are four distinct types of faults: (1) simple configuration errors (*e.g.*, typos); (2) semantic configuration errors (*e.g.*, misunderstood effects of parameters); (3) broken environmental dependencies (*e.g.*, library or port conflicts); and (4) data-access errors, which render the persistent data partially unavailable. I also estimate that, on average, Type 1 faults occur in 14.6% of upgrades, and Type 4 faults occur in 18.7% of upgrades. Understanding deployment-time dependencies in distributed systems and the impact of operator errors during software upgrades is the first step toward an approach for masking the common upgrade faults and improving the system dependability.

Challenge and Contributions

While Crameri et al. [2007] suggest that most upgrades fail by breaking dependencies, the failure mechanisms and their effects are not well understood. Our current knowledge of upgrade faults is largely anecdotal, stemming from known examples of failed upgrades (see Section 1.1), which do not provide sufficient data for replicating the failure. Real-world data on upgrade-faults is scarce and hard to obtain due to the sensitivity of this subject.

To improve the dependability of software upgrades, we must understand what upgrade failures have in common and how to avoid these hazards. Addressing this challenge will

require answering several open questions: How many distinct types of upgrade faults are there? How often do upgrade faults occur in practice? What type of fault is most likely to cause upgrade failures? How effective are the existing mechanisms in detecting and tolerating upgrade faults? The goal of this chapter is to *establish an upgrade-centric fault model* that can serve as the basis for dependable upgrade mechanisms (Chapter 6) and for benchmarking the dependability of software upgrades (Chapter 7).

Previous attempts at classifying upgrade faults [for example: Oppenheimer et al., 2003; Nagaraja et al., 2004; Oliveira et al., 2006; Keller et al., 2008] did not produce rigorous fault models because the fault categories are not disjoint, the criteria for establishing these categories remain unclear, and the classifications are relevant only for subsets of the upgrade faults. I analyze 55 upgrade faults from the best available sources, and, through statistical cluster-analysis, I establish four categories of upgrade faults.

Assumptions. I combine data from three independent studies on operator errors, collected using different experimental methods. While the operators targeted by these studies focus on different problems and handle different workloads, I start from the premise that they use similar mental models during change-management tasks, which yield comparable faults. This hypothesis is supported by the observation that several faults have been reported in more than one study. Furthermore, as each of the three studies is likely to emphasize certain kinds of faults over others, I provide a better coverage of upgrade faults than previous studies.

Non-goals. The upgrade-centric fault model focuses on procedural errors rather than software defects. I exclude software defects from the classification because they have been rigorously classified before [Sullivan and Chillarege, 1991; Lu et al., 2005] and because most upgrade failures are not due to software defects.

This chapter makes four contributions:

- I identify the presence of *hidden dependencies* in distributed systems and I show that breaking such hidden dependencies currently represents leading cause of upgrade failures.
- I establish a *rigorous classification of upgrade faults*, with four distinct categories. This classification represents an upgrade-centric fault model.

- I estimate the *range of the fault frequencies* for two of the four types of upgrade faults, with high statistical confidence.
- I show that two of the four types of upgrade faults are *not adequately addressed by existing techniques*.

Section 3.1 describes the criteria used for classifying upgrade faults and the three data sources. Section 3.2 presents the upgrade-centric fault model. Section 3.3 discusses the applicability of existing techniques for tolerating the upgrade-specific faults.

3.1 Classification method

I classify upgrade faults recorded in three independent sources. I conduct a *field study* of bug reports filed in 2007 for the Apache web server [Dumitraș et al., 2008]. I combine these reports with data collected by other researchers: a *user study* of system-administration tasks in an e-commerce system [Nagaraja et al., 2004] and a *survey* of database administrators [Oliveira et al., 2006]. Because each of the three methods is likely to emphasize different kinds of faults, combining these dissimilar data sets allows me to provide a better coverage of upgrade faults than previous studies. I classify these upgrade faults using *statistical cluster analysis*, which is widely used in the natural sciences for establishing taxonomies of living organisms.

I select the data points and classification features to include in the fault model according to the following three criteria:

C1 *Hardware and software defects are orthogonal to the upgrading concerns.* While some upgrades fail because of software defects [Crameri et al., 2007], these defects occur for reasons that are not related to the upgrade and they might be exposed in other situations as well. The programmers who introduce the software defects are not usually involved in upgrade procedures, and they have different mindsets from the system administrators who execute the upgrades. Similarly, hardware defects occur for reasons unrelated to the upgrade and, therefore, I exclude them from the fault model.

C2 *I classify upgrade faults, not their impacts.* The fault impact depends not only on the upgrade fault, but also on the system architecture. For instance, if the system em-

ploys replicated servers, a fault that disables one server may be masked owing to this redundancy. I do not include the fault impact among the classification features in order to avoid establishing a connection among distinct faults, which occur in different ways, but which have similar outcomes.

C3 *I classify upgrade faults, not upgrade tasks.* Similarly, an upgrade fault can occur during different upgrade tasks. To avoid establishing a connection among multiple faults for the sole reason that they were recorded during the same task, I exclude the task from among the classification features.

3.1.1 Sources of fault data

Field study. I analyze bug reports for the Apache 2.2 web server, filed between 1 January 2007 and 21 December 2007 [Dumitraș et al., 2008]. I focus on closed reports that have been marked `INVALID` or `WONTFIX`, which usually indicate configuration or procedural errors. These errors have serious impacts, which prompted the opening of a bug report, but they have been excluded from the previous studies of the Apache bug database [for example: Li et al., 2006; Kim et al., 2007], which focus on classifying software defects. I determine a preliminary classification of these faults, which yields seven categories: build, paths and permissions, environmental conflicts, third-party error, parameter tuning, syntax error, semantic error. I try to determine which faults occur during upgrade-related tasks by searching for keywords such as "upgrade," "update" or "install."

User study. Nagaraja et al. [2004] conduct a user study with 21 system administrators, with varying degrees of experience, who are asked to perform several system-administration tasks. The authors of the study observe 32 instances of 16 unique faults and 10 misdiagnoses; 25 instances of 13 unique faults occur during upgrade-related tasks. The study identifies seven classes of faults; the most frequent faults are "global misconfigurations," which compromise the communication between different components of the system.

Survey. Oliveira et al. [2006] conduct a survey of 51 database administrators. The DBAs report that their regular-maintenance tasks are related to change management (*e.g.*, tuning the performance, changing the database structure, modifying the data, coding and upgrading the software), to runtime monitoring (*e.g.*, space monitoring/management, system

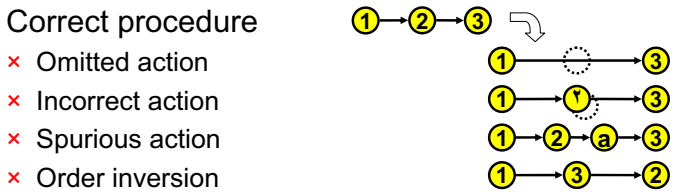


Figure 3.1. Four ways of violating an upgrade procedure. From top to bottom, the procedure violations are ordered by how frequently they are recorded in the upgrade-fault data.

monitoring, integrity checks, performance monitoring), and to recovery preparations (*e.g.*, making/testing backups, conducting recovery drills). The survey identifies nine categories of problems faced by DBAs and provides details for 23 distinct problems.

3.1.2 Classification features

I conduct a *post-mortem* analysis of each fault from the three studies in order to determine its root cause [Oppenheimer et al., 2003]—*configuration* error, *procedural* error, *software* defect, *hardware* defect—and whether the fault has broken a hidden dependency, with repercussions for several components of the system-under-upgrade. Errors introduced while editing configuration files can can be further subdivided in three categories [Keller et al., 2008]: *typographical errors* (typos), *structural errors* (*e.g.* misplacing configuration directives), and *semantic errors* (*e.g.* ignoring constraints among configuration parameters). Additionally, a small number of configuration errors do not occur while editing configuration files (*e.g.*, setting incorrect access privileges). Operators can make procedural errors by performing an *incorrect action* or by violating the sequence of actions in the procedure through an *omission*, an *order inversion*, or the addition of a *spurious action* [Dumitraș and Narasimhan, 2009a]. These procedure violations are illustrated in Figure 3.1.

Because the upgrade faults correspond to human errors, I also take into consideration the cognitive level involved in the error, While I do not classify the faults based on the tasks where they occur (according to Criterion C3), the corresponding cognitive levels allow me to distinguish between faults that occur during simple and complex procedures. There are three cognitive levels at which humans solve problems and make mistakes [Reason, 1990]: the *skill-based* level, used for simple, repetitive tasks, the *rule-based* level, where problems are solved by pattern-matching, and the *knowledge-based* level, where tasks are approached by reasoning from first principles. For example, I consider that all path-related faults occur

on the rule-based cognitive level and that all the faults related to database schemata occur on the knowledge-based level. I consider that faults related to file-system and database access-privileges are configuration errors, even if they don't involve editing configuration files, and they occur on the rule-based cognitive level.

3.1.3 Statistical cluster analysis

I compare the upgrade faults using the *Gower distance* [Gower, 1971], based on the categorical values of five classification features: (i) the root cause of each fault; (ii) the hidden dependency that the fault breaks (where applicable); (iii) the fault location—*front-end*, *middle-tier*, or *back-end*—; (iv) the original classification, from the three studies, which encodes the domain knowledge relevant to the fault; and (v) the cognitive level involved in the reported operator error (see also Table 3.1). The high-level fault descriptions from the three studies are sufficient for determining the values of the five classification variables. I perform agglomerative, hierarchical clustering with *average linkage* [Kaufman and Rousseeuw, 1990].

I include in the classification all the faults reported in the three studies, except for software defects, faults that do not occur during upgrades and client-side faults. I exclude hardware and software defects[1] from the taxonomy, according to criterion $\mathbb{C}1$.

It is interesting to note that some faults are reported in more than one source. For example, a configuration error (`apache_config_staticpath`) where Apache is instructed to serve static HTML files from an existing, but incorrect, location is reported in both the user and field studies. A configuration error (`wrong_privileges_insufficient`) where the application is granted insufficient access privileges to database tables is reported in both the user study and the survey. A procedural error (`wrong_apache`) where the wrong version of the Apache web server is started in the front-end is reported during three different tasks, executed by different operators, in the user study. Another procedural error (`db_schema_mismatch`), where the application queries an incorrect database schema, is reported in conjunction with two different tasks in the survey. A configuration error (`apache_largefile`) which prevents Apache from sending files larger than 64K, is the source of three bug reports in the field study.

[1] The fault descriptions provided in the three studies allow me to distinguish the operator errors from the manifestations of software defects.

Table 3.1. Classification features for upgrade faults

Variable	Categories	Description
Location [Oppenheimer et al., 2003]	Front-end	Fault in the front-end of the infrastructure, which handles the client connections.
	Middle tier	Fault in the middle tier of the infrastructure, which processes client requests.
	Back-end	Fault in the back-end of the infrastructure, where persistent data is stored (typically, in a database).
Root cause [Oppenheimer et al., 2003; Keller et al., 2008]	Software	Software defect.
	Hardware	Failure of a hardware component.
	Configuration	Operator error while configuring the system. Errors in configuration files can be categorized as *typographical*, *structural* or *semantic*.
	Procedure	Procedural error. Can be a *incorrect action*, an *omission*, an *order inversion* or a *spurious action*.
Hidden dependency	see Table 3.2	Hidden-dependency broken (where applicable).
Original classification:	Global misconfiguration	Configuration inconsistencies compromising the communication between system components.
	Local misconfiguration	Configuration error affecting a single node.
	Start of wrong version	Configuring one version and starting a different version of a software component.
from user study [Nagaraja et al., 2004]	Unnecessary restart	Unnecessarily restarting a software component.
	Incorrect restart	Starting a software component incorrectly (*e.g.*, without the necessary access privileges).
	Unnecessary replacement	Misdiagnosing the problem as a hardware fault.
	Wrong hardware	Installing the database on a slow disk.
	Deployment	Changes to the online system (previously tested offline) cause the database to misbehave.
	Performance	The DBMS delivers poor performance.
from survey [Oliveira et al., 2006]	General structure	Incorrect database design or unsuitable schema.
	DBMS	Software defects in the DBMS.
	Access privileges	Insufficient or excessive access privileges granted to users or applications.
	Space	Disk space or tablespace exhaustion.
	General maintenance	Other problems (*e.g.* incompatible upgrades, incorrect restarts).
	Hardware	Hardware failure and potential data loss.
	Build	Missing shared libraries prevent compilation.
	Paths and permissions	Incorrect paths or insufficient access permissions.
from field study [Dumitraș et al., 2008]	Environmental conflicts	Wrong library versions, byte orders, *etc.*
	Third-party error	Software defects or misconfigurations in third-party components.
	Parameter tuning	Incorrect setting for a parameter.
	Other error	Other errors in the application's configuration file (*e.g.*, missing or incorrect commands, wrong order of commands, typos, syntax errors).
Cognitive level [Reason, 1990]	Skill-based	Slips and lapses during common, repetitive tasks.
	Rule-based	Mistakes when reasoning and solving problems through pattern matching.
	Knowledge-based	Mistakes when reasoning from first principles.

In the cluster analysis the distance between faults reported multiple times is 0. To avoid placing identical faults in different clusters, I merge their original classifications, coming from different studies. This pre-processing step merges the "access-privilege problems" from the survey with the "global misconfigurations" from the user study, and the "path and permissions" classification from the field study with the "local misconfigurations" from the user study.

I consider that faults occurring in the same way (*e.g.*, by configuring the wrong port for a server), but which are located on different tiers, do not represent instances of the same fault, because they are likely to be introduced by different kinds of operators (*e.g.*, system administrators, database administrators, application maintainers) who use different mental models for the tasks they perform. The annotated faults used in the classification are listed in Appendix B, and they can also be downloaded from `http://www.ece.cmu.edu/~tdumitra/upgrade_faults/`.

3.2 Upgrade-centric fault model

The most frequent procedure violations (see Figure 3.1) are omissions, which account for 22% of all procedural errors, followed by incorrect actions, which account for 15% of all procedural errors. In 56% of these cases, however, the operators introduce upgrade faults despite correctly following the mandated procedure.[2] This suggests that procedure violations alone cannot explain the occurrence of upgrade faults.

Distributed systems often include *hidden dependencies*—dependencies that cannot be detected automatically (*e.g.* because they only manifest transiently, at runtime) or that are overlooked because of their complexity (*e.g.* dependencies on configuration settings). 85% of all the configuration and procedural errors in the data break hidden dependencies. Table 3.2 lists the observed hidden dependencies. For example, the location of a service (*e.g.*, the path to a local file or the network address of a server) is often specified incorrectly during an upgrade. Conflicts between dynamically-linked libraries usually occur in spite of following the correct upgrading procedure. In some cases, when the upgrade effects

[2]The remaining procedural errors are caused by avoidable human mistakes, *e.g.* forgetting to modify a configuration file altogether.

changes to the database schema, the schema queried by middle-tier servers does not match the schema materialized in the back-end database.

Incorrect or omitted actions sometimes occur because the operators ignore, or are not aware of, certain dependencies among the system components Conversely, some procedure violations introduce hidden dependencies, *e.g.* when the value of a parameter must be specified twice during an upgrade procedure (and one of these actions is omitted or performed incorrectly) or when an order inversion causes the replication degree of a tier (*e.g.* the database in the back-end) to drop to 0, rendering the service unavailable.

This illustrates the fact that even well-planned upgrades can fail because the complete set of dependencies is not always known in advance. I emphasize that the list of hidden dependencies from Table 3.2, obtained through a *post-mortem* analysis of upgrade faults, is not exhaustive and that other hidden dependencies might exist in distributed systems, posing a significant risk of failure for distributed-system upgrades.

3.2.1 The four types of upgrade faults

Cluster analysis (Figure 3.2) suggests that there are four natural types of faults:

- **Type 1** corresponds to simple configuration errors (typos or structural) and to procedural errors that occur on the skill-based cognitive level. These faults break dependencies on network addresses, file paths, or the replication degree.
- **Type 2** corresponds to semantic configuration errors, which occur on the knowledge-based cognitive level and which indicate a misunderstanding of the configuration directives used. These faults break dependencies on the request scheduling, cached data, or parameter constraints.
- **Type 3** corresponds to broken environmental dependencies, which are procedural errors that occur on the rule-based cognitive level. These faults break dependencies on shared libraries, listening ports, communication protocols, or access privileges.
- **Type 4** corresponds to data-access errors, which are complex procedural or configuration errors that occur mostly on the rule- and knowledge-based cognitive levels. These faults prevent the access to the system's persistent data, breaking dependencies on database schemata, access privileges, the replication degree, or the storage availability.

Table 3.2. Examples of hidden dependencies that affect distributed-system upgrades (sorted by their frequency of occurrence).

Hidden dependency	**Procedure violation**	**Impact**
Service location: • File path • Network address	Omission	Components unavailable, latent errors
Dynamic linking: • Library conflicts • Defective 3rd party components		Components unavailable
Database schema: • Application/database mismatch • Missing indexes	Omission Omission	Data unavailable Performance degradation
Access privileges to file system, database objects, or URLs: • Excessive • Insufficient • Unavailable (from directory service)	Wrong action Omission Omission	Vulnerability Components/data unavailable
Constraints among configuration parameters		Outage, degraded performance, vulnerability
Replication degree (*e.g.*, number of front-end servers online)	Omission, inversion, spurious action	Outage, degraded performance
Amount of storage space available	Omission	Transactions aborted
Client access to system-under-upgrade	Wrong action	Incorrect functionality
Cached data (*e.g.*, SSL certificates, DNS lookups, kernel buffer-cache)		Incorrect functionality
Listening ports	Omission	Components unavailable
Communication-protocol mismatch (*e.g.*, middle-tier not HTTP-compliant)		Components unavailable
Entropy for random-number generation		Deadlock
Request scheduling		Access denied unexpectedly
Disk speed	Wrong action	Performance degradation

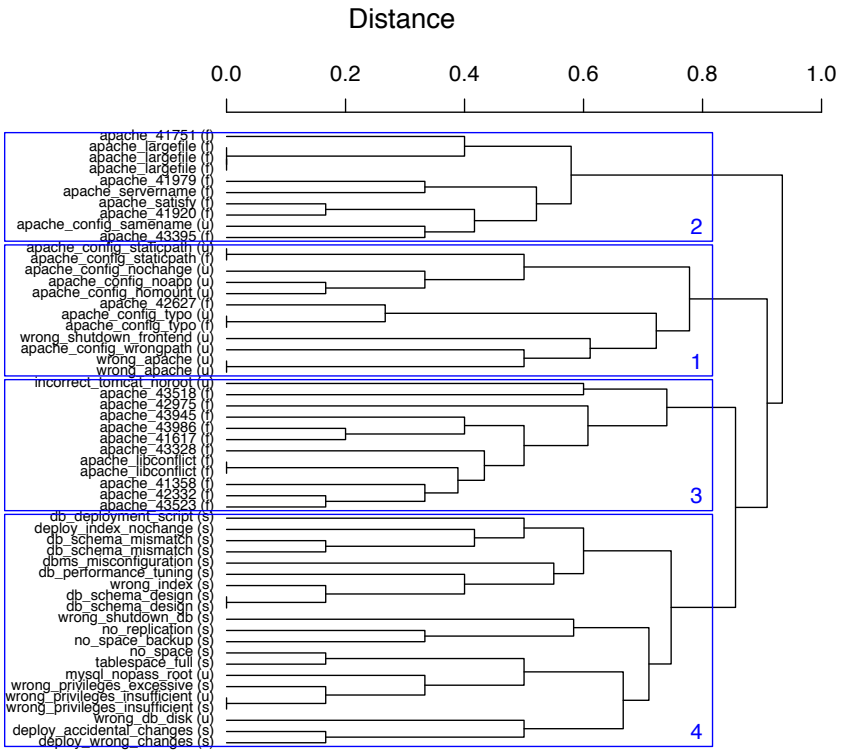


Figure 3.2. Statistical cluster analysis of upgrade faults. In this clustering *dendrogram* the leaves correspond to the 55 faults reported in the field study (f), the user study (u) and the survey (s). Each vertical line links two clusters into a larger cluster, and their position on the X-axis indicates the mean inter-fault distance. A link with a significantly larger distance than the links below suggests the presence of a natural cluster (highlighted by a blue rectangle). The *cophenetic correlation coefficient*, which shows the correlation between the inter-fault distance and the distance in the dendrogram, is 0.85.

Faults that occur while editing configuration files are of type 1 or 2. Types 1–3 are located in the front-end and in the middle tier, and, except for a few faults due to omitted actions, they usually do not involve violating the mandated sequence of actions. Type 4 faults occur in the back-end, and they typically consist of omissions or incorrect actions.

I use *principal component analysis* [Kaufman and Rousseeuw, 1990] to determine if the four clusters overlap. This statistical technique reduces the dimensionality of the upgrade-fault data from the five classification features (described in Section 3.1.2) to two principal components, which are plotted in Figure 3.3a. Principal-component analysis suggests that the four types of faults correspond to disjoint and compact clusters. The values of the first principal component, which corresponds to the x-axis in Figure 3.3a, indicate if the faults are procedural errors or if they that occur while editing configuration files. The second principal component (the y-axis) corresponds, approximately, to the hidden dependencies

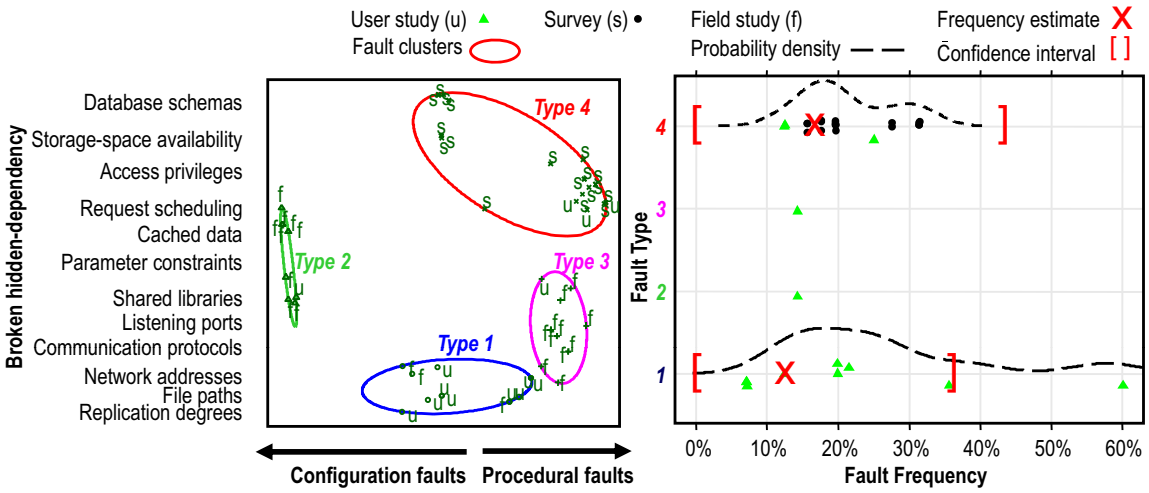


Figure 3.3. Upgrade-centric fault model. Principal-component analysis (a) creates a two-dimensional shadow of the five classification variables. The survey and the user study also provide information about the distribution of fault-occurrence rates (b).

broken by the upgrade faults.

3.2.2 Frequency of upgrade faults

I also estimate how frequently these fault types occur during an upgrade (see Figure 3.3b), by considering the percentage of operators who induced the fault (during the user study) or the percentage of database administrators who consider the specific fault among the three most frequent problems that they have to address in their respective organizations (in the survey). I cannot derive frequency information from the field-study data. The individual estimations are imprecise, because the rate of upgrades is likely to vary among organizations and administrators, and because of the small sample sizes (5–51 subjects) used in these studies. I improve the precision of our estimations by combining the individual estimations for each fault type.[3]

I estimate that Type 1 faults occur in 14.6% of upgrades (with a confidence interval of $[0\%, 38.0\%]$ significant at the $p = 0.05$ level). Most Type 1 faults (recorded

[3]The *precision* of a measurement indicates if the results are repeatable, with small variations, and the *accuracy* indicates if the measurement is free of bias. While in general it is not possible to improve the accuracy of the estimation without knowing the systematic bias introduced in an experiment, minimizing the sum of squared errors from dissimilar measurements improves the precision of the estimation [Chatfield, 1983].

in the user study) occur in fewer than 21% of upgrades, with one exception: fault `apache_config_staticpath`, which has an occurrence frequency of 60%. However, this fault was recorded in an experiment using a sample of only 5 subjects, and such a small sample leads to an imprecise estimation, reflected in the fact that the `apache_config_staticpath` fault does not lie within the confidence interval computed for Type 1 faults.

Similarly, I estimate that Type 4 faults occur in 18.7% of upgrades (with a confidence interval of $[0\%, 45.1\%]$ significant at the $p = 0.05$ level). Because faults of types 2 and 4 are predominantly reported in the field study, we lack sufficient information to compute a statistically-significant fault frequency for these clusters.

It is interesting to compare these numbers with the failure rates previously reported in the literature. Unlike the previous studies, I focus on characterizing upgrade *faults*, which correspond to the common root causes of upgrade problems and which might not lead to system failure because they are masked or tolerated. Crameri et al. [2007] report that, on average, 8.6% of upgrades fail, with a maximum failure rate of 50%. This is consistent with upgrade-fault rates of up to 45% estimated in this section.

3.2.3 Impact of upgrade faults

An upgrade has *failed* if its outcome is unacceptable, *e.g.* it rendered the system unavailable or it caused data corruption. In these cases, the upgrade must be rolled back or further changes must be implemented to bring the system into a correct operating mode. However, it is difficult to determine with certainty when an upgrade has failed from the fault descriptions provided in the three studies. For the field study, in particular, the fault impacts cannot always be inferred from the terse bug reports. Moreover, because upgrades are sometimes executed during maintenance windows, when the system is offline or when it operates in a mode with degraded functionality (*e.g.*, some services are not available or the system's throughput is reduced), we must distinguish between such expected degradations and upgrade failures. I therefore make a conservative estimation of the upgrade-failure rates by considering only the fault impacts that affect the system's clients and that cannot constitute acceptable outcomes for an upgrade. I include software defects in this comparison, but I consider only bugs that reportedly affect upgrade-related tasks. A fault

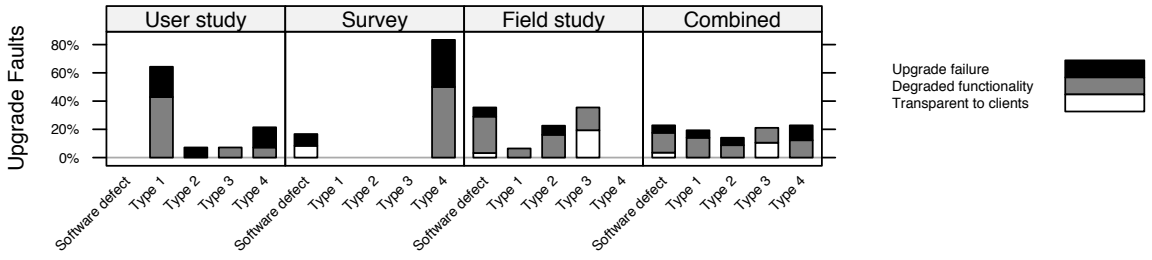


Figure 3.4. Impact of upgrade faults.

may have more than one impact; for example `wrong_apache` causes an outage if it affects all the front-end servers and a throughput degradation if it affects only a few.

I consider that the upgrade has failed when faults cause unplanned downtime or incorrect functionality, or when they introduce security vulnerabilities or latent errors. These faults cannot be masked by enhancing the system's architecture, *e.g.* by increasing the component redundancy. Fault impacts such as increased end-to-end latency or reduced throughput constitute degraded functionality, which could be acceptable during an upgrade. Other faults affect only server-side functions (*e.g.* logging), but have no client-visible impact; we consider that these faults do not induce upgrade failures. The actual number of upgrade failures is therefore higher than the number of certain failures and lower than the number of faults that are not transparent to the clients.

In the three studies, latent errors are introduced only by Type 1 faults. Types 2 (in the field study) and 4 (in the survey) lead to incorrect functionality or security vulnerabilities. All four fault types, as well as software defects, may cause outages, throughput degradation or increased latencies. Type 3 faults can also produce compilation errors, due to broken environmental dependencies such as missing libraries (we consider that this impact is transparent to the clients because it occurs during the preparation phase of the upgrade). The software defects and upgrade faults reported in the three studies do not cause loss of data. The most common impact of upgrade faults is throughput degradation.

Figure 3.4 compares the outcomes of upgrades affected by software defects and upgrade faults. Type 1 and 2 faults and software defects have similar impacts, each causing 20% of the recorded upgrade failures. The leading cause of upgrade failures seem to be Type 4 faults, which are responsible for 40% of failures.[4]

[4]This cannot be verified in the field study, where the succinctness of the bug reports leads to fewer indisputable failures.

3.2.4 Threats to validity

Each of the three studies has certain characteristics that might skew the results of the cluster analysis. For example, because the user study is concerned with the behavior of the operators, it does not report any software defects or hardware failures. Configuration errors submitted as bugs tend to be due to significant misunderstandings of the program semantics, and, as a result, our field study contains an unusually-high number of faults occurring on the knowledge cognitive level. Moreover, the results of bug searches are not repeatable because the status of bugs changes over time; in particular, more open bugs are likely to be marked as `INVALID` or `WONTFIX` in the future. Finally, Crameri et al. [2007], who identify broken dependencies as the leading cause of upgrade failures, caution that their survey is not statistically rigorous.

3.3 Tolerating upgrade faults

Industrial best-practice recommendations [for example: Office of Government Commerce, 2007] place a significant focus on the upgrade procedure and the ordering of mandated actions. However, the empirical data presented here shows that procedure violations account for less than half of the upgrade faults recorded. This suggests that most upgrade faults are due to unavoidable human errors that break hidden dependencies in the system-under-upgrade.

Several automated dependency-mining techniques have been proposed [for example combining static and semantic analysis: Dig et al., 2006; see also Section 1.1.1], but these approaches cannot provide a complete coverage of dependencies that only manifest dynamically, at runtime. The upgrade-centric fault model introduced in this chapter emphasizes the fact that different techniques are required for tolerating each of the four types of faults. For example, modern software components check the syntax of their configuration files, and they are able to detect many Type 1 faults at startup (*e.g.*, syntax checkers catch 38%–83% of typos [Keller et al., 2008]).

Type 2 faults are harder to detect automatically. Keller et al. [2008] argue that tracking the dependencies among the values of configuration parameters can improve the robustness against such semantic faults, while Zheng et al. [2007] show how to generate configurations that tune a specific metric (*e.g.* server-side throughput) by solving a constrained-

optimization problem. To prevent the faults that fall under Type 3, modern operating systems include package management tools that use dependency-tracking to upgrade a software component along with all of its dependencies [Tucker et al., 2007; Di Cosmo et al., 2008]. Best practices in distributed-system administration recommend extending this approach to maintain all the dependency information in the system [Office of Government Commerce, 2001] in a centralized Configuration Management Database (CMDB). However, this would require existing package-management frameworks to handle additional information (*e.g.* configuration dependencies), which would increase the size of their databases by at least one order of magnitude. The size of a complete CMDB would likely expose the fundamental limitations of dependency tracking (see Section1.1.1).

We currently lack automated techniques for handling Type 4 faults. Oliveira et al. [2006] propose validating the actions of database administrators using real workloads, which prevents some Type 4 faults, but they also remark that this is difficult to implement when the administrator's goal is to change the database schema or the system's observable behavior.

For these reasons, practitioners often prefer to deploy the new version gradually, in successive stages [Office of Government Commerce, 2007; Downing, 2008]. This is commonly implemented through a rolling upgrade [Brewer, 2001; see also Section 2.3.4] , which upgrades and then reboots each node, in a wave rolling through the distributed system. Rolling upgrades require old and new versions of the system to interact with the data store and with each other in a compatible manner. However, current commercial products for rolling upgrades [for example: Microsoft Corporation, 2005; Oracle Corporation, 2008] provide no way of determining if the interactions between mixed versions are safe and introduce the risk of breaking hidden dependencies.

Existing techniques aim to detect or prevent upgrade faults (*e.g.*, through dependency tracking), rather than masking them from the clients. While most Type 1 faults can be detected in this manner, and there is anecdotal evidence that the severity of Type 3 faults is declining due recent improvements in package management tools, we currently lack efficient mechanisms for tolerating upgrade faults of Types 2 and 4. Of all the upgrade faults analyzed in this chapter, 85% break dependencies that remain hidden from the operators performing the upgrade (see Table 3.2), and many upgrade faults occur despite correctly following the upgrading procedure. This suggests that a novel approach is needed for improving the dependability of enterprise-system upgrades.

3.4 Summary of findings

I establish an upgrade-centric fault model, by analyzing data from three independent sources [Dumitraș and Narasimhan, 2009a]: (i) a user study of system administration tasks in an e-commerce system, (ii) a survey of database administrators and (iii) a field study of bug reports for the Apache web server. As each of the three studies is likely to emphasize certain kinds of faults over others, combining these dissimilar data sets allows me to provide a better coverage of upgrade faults than previous studies. My fault model emphasizes unavoidable human errors in the upgrade procedure that break hidden dependencies—*e.g.*, service locations specified incorrectly, shared-library conflicts, database-schema mismatches between the middle tier and the back-end, conflicts—in the system-under-upgrade.

There are four types of faults the commonly occur during software upgrades: (1) simple configuration errors (*e.g.*, typos); (2) semantic configuration errors (*e.g.*, misunderstood effects of parameters); (3) broken environmental dependencies (*e.g.*, library or port conflicts); and (4) data-access errors, which render the persistent data partially unavailable. Faults of types 1 and 4 occur in up to 45% of upgrades.

Existing upgrade mechanisms rely on tracking the dependencies among system components, but they are reaching their limit owing to the increasing complexity of configuration dependencies and to the presence of dynamic dependencies that cannot always be discovered automatically. In order to improve the reliability of distributed systems, we must develop upgrade mechanisms that function correctly despite hidden dependencies or that are able to tolerate Types 1–4 of upgrade faults.

This will require downtime on upgrade, so we're not going to do it until we have a better idea of the cost.

MediaWiki revision-control log, 2004

Chapter 4

Why Do Upgrades Need Planned Downtime?

UNAVAILABILITY in distributed enterprise systems is usually the result of planned events, such as software upgrades, rather than component failures. Chapter 3 has shown that upgrades are unreliable and often result in unplanned downtime. Moreover, even successful upgrades may require downtime, and as a result they are performed offline, during windows of planned maintenance. A survey of 426 sites with high-availability applications, using servers from IBM, Sun, Compaq, HP, DEC, and Tandem, showed that 75% of nearly 6,000 outages were due to planned hardware and software maintenance and that such planned outages typically lasted twice as long as unplanned ones [Lowell et al., 2004]. This chapter focuses on the common causes of such *planned downtime*.

Many enterprises can no longer afford to incur the high cost of downtime [Patterson, 2002] and must perform such upgrades online, without stopping their systems. For example, upgrading enterprise resource planning (ERP) systems can cost up to 30% of the price of the original implementation ($40M – $240M) [Beatty and Williams, 2006]. Experience with AT&T network upgrades suggests that there are no good time windows for scheduling planned maintenance; for instance, performing network maintenance at night is likely to disrupt online gaming [Rexford, 2007].

Enterprise-system upgrades often require complex data conversions for changing the data schema or migrating to a different data store. Synchronizing the states of two system versions during an in-place upgrade requires developers and administrators to define state-transfer functions and to ensure the correctness of mixed-version interactions [Ajmani et al., 2006; Oracle Corporation, 2008]. Administrators working on complex systems, which have dependencies that are not fully understood, also fear that online upgrades could loose or

corrupt persistent data [Williams, 2009].

Moreover, because some data conversions are difficult to perform on the fly, in the face of live workloads, and owing to concerns about overloading the production system, *upgrades that involve computationally-intensive data conversions currently necessitate planned downtime*, ranging from tens of hours to several days [Downing, 2008]. Conversely, system administrators sometimes avoid complex upgrades, which might impose an unacceptable downtime, and preserve database schemas that provide sub-optimal performance and that cannot support new, user-requested features.

Challenge and Contributions

Research on software upgrades has been conducted independently, in separate research communities, and has focused on upgrading individual components of distributed systems, such as the application code, the middleware framework or the database schema. The prior work on database schema evolution [for example: Ferrandina et al., 1995; Bernstein and Haas, 2008; Curino et al., 2008a] focused on automating the process of defining the mapping between the old version and the new version of the schema. However, which schema changes are difficult to integrate in an online end-to-end upgrade and, in general, what are the leading causes of planned downtime remains an open question.

Assumptions. In my study, I focus on real-world distributed systems, and I assume that upgrades are performed in-place, replacing the existing system. These systems are not based on a homogeneous middleware framework. Instead, they utilize three-tier architectures, with front-end servers that manage the client connections, middle-tier servers that implement the application logic and back-end servers that store the persistent data. Furthermore, the results presented in this chapter derive from a single case study—which focuses on an Internet service that is representative for three-tier enterprise systems and that accommodates reproducible research—, but they are correlated with observations about planned downtime in several commercial systems.

Non-goals. In this study, I do not focus on upgrades that cause unexpected downtime, which is the subject of Chapter 3. Moreover, I focus on the technical challenges that impose planned downtime when upgrading. I do not consider downgrades or other activities performed during planned-maintenance windows.

This chapter makes two contributions:

- By analyzing the upgrade history of Wikipedia, one of the ten most popular websites to date (Section 4.1), I identify the common causes of planned downtime (Section 4.2).
- I show that the existing mechanisms for online upgrade do not handle these causes of downtime effectively (Section 4.3).

4.1 Experimental method

I determine the common reasons for planned downtime by studying in depth the upgrade history of Wikipedia, which is currently is one of the ten most popular Internet services.[1] Wikipedia uses open-source software, the system architecture and the individual host configurations are known, the workload has been analyzed rigorously and the content of the database is available for download. Because of these characteristics, Wikipedia represents a testbed well suited for conducting repeatable research on distributed-system management. Moreover, because Wikipedia is used widely, it relies of a typical three-tier enterprise architecture and its software is employed by other systems as well, the causes of upgrade-related downtime at Wikipedia are representative of the root causes of planned downtime in distributed enterprise systems.

I study the upgrade history of Wikipedia by combining data from a rigorous study of Wikipedia's schema evolution with information from design documents and archived discussions [Dumitraș and Narasimhan, 2009b]. I also correlate the findings from this study with anecdotal and private information about downtime in three commercial systems.

Wikipedia. The site `http://www.wikipedia.org` provides a multi-language, free encyclopedia, handling peak loads of 70,000 HTTP requests/s.[2] The English-language Wikipedia has 2.9 million articles, with content stored in a 1 TB database and 250,000 files (*e.g.* images). The web site is supported by an infrastructure (Figure 4.1) running on 352 servers in 3 data centers distributed worldwide. The front-end has 120 caching proxies, accessed using round-robin DNS load-balancing. The proxies forward the cache-misses to a load-balanced cluster of 206 Apache web servers, which query 23 MySQL database

[1] Statistics on the popularity and workloads of different web sites are available at `http://www.alexa.com`. Wikipedia has been on the top ten list since 2007.

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

servers, configured for master/slave replication. The master database receives the write queries and propagates the updates to the slave, which handle read-only queries. A wiki engine called MediaWiki, implemented as a set of PHP scripts, accesses the database and generates the content of the articles.

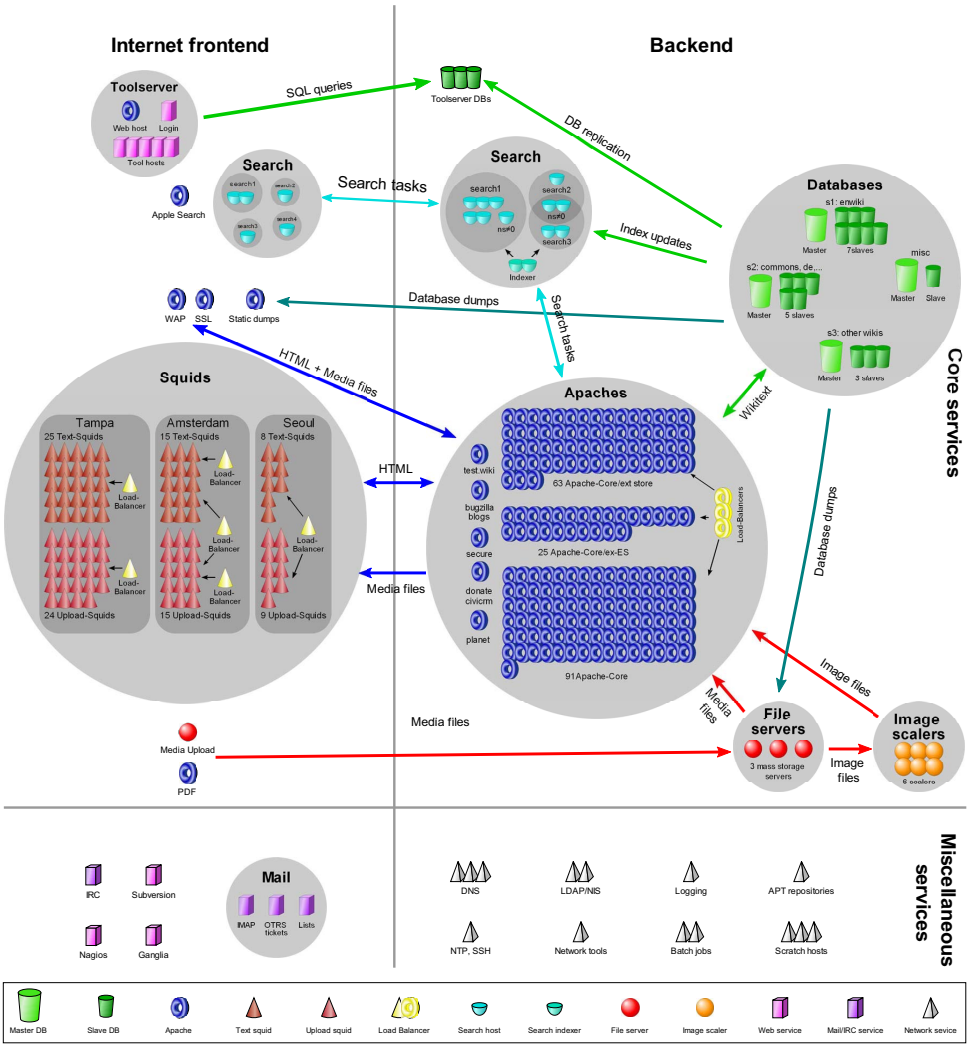
All the data needed to reproduce Wikipedia's infrastructure is publicly available:

- Wikipedia uses only open-source software, which is employed by other systems as well. For example, MediaWiki drives over 15,000 other wikis (statistics on the usage of wiki engines are maintained at `http://s23.org/wikistats/`).
- Wikipedia relies on a standard three-tier architecture, and its hardware and software configurations are described at `https://wikitech.leuksman.com`.
- The database schema employed by Wikipedia can be recreated using the installation scripts of MediaWiki, and the database content can be downloaded from `http://download.wikimedia.org/`.
- The workload characteristics are recoded at `http://en.wikipedia.org/wiki/Wikipedia:Statistics`, and they have been rigorously characterized by Urdaneta et al. [2009].

Between 2003 and 2008, MediaWiki's database schema has gone through 171 evolution steps [Curino et al., 2008b] in the main development branch. During this time, the project has released eleven versions (1.1 – 1.11) of the wiki engine. Minor versions (*e.g.*, the 1.4.x series) do not introduce schema changes; upgrading to a new version within the same release series requires only replacing the PHP code on the application servers. However, upgrading to a different major version (*e.g.* from 1.4 to 1.5) can require changes in the database. The remainder of this section describes the techniques employed by Wikipedia for upgrading online and provides examples of database changes that are not handled adequately by these techniques and that impose planned downtime.

4.1.1 Current approaches for online upgrade at Wikipedia

In the early days of the site, when Wikipedia relied on a single database server in the backend, downtime was hard to avoid during an upgrade. In some situations, it is possible to allow read-only access concurrently with the upgrade procedure. MediaWiki provides a configuration parameter (`$wgReadOnly`) that places the site in read-only mode for such planned-maintenance activities.



Source: `http://meta.wikimedia.org/wiki/Wikimedia_servers`

Figure 4.1. Wikipedia architecture.

If the database is replicated on multiple hosts, it is possible to employ a rolling upgrade [Brewer, 2001; see also Section 2.3.4] in order to avoid planned downtime. At Wikipedia, a rolling upgrade removes slave nodes one by one from the replication group, applies the schema changes, and then restarts the replication. The rolling upgrade swaps database masters before completing the schema upgrade, to avoid re-applying the changes through the replication mechanism.

To support rolling upgrades, the database replication mechanism must allow source and target tables that do not have identical definitions. For example, in MySQL, a table on the master node can have more or fewer columns than the slave's copy, or a column on the

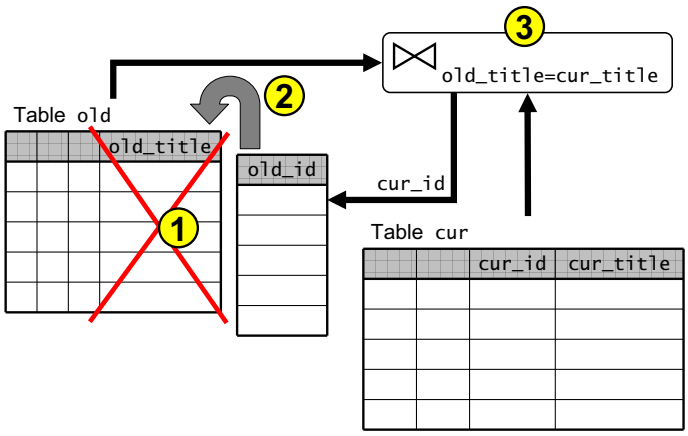


Figure 4.2. Example of schema change that can impose planned downtime: a column ① must be replaced with another one ② and initialized with data from a different table ③.

slave node can use a larger storage size than the corresponding column on the master (the two columns must have the same data type, *e.g.* `INT(`*storage size*`)`). In general, rolling upgrades require the new version to be backward-compatible [Brewer, 2001], which excludes complex schema changes that would prevent the old version of the business logic to query the new version of the database schema.

4.1.2 Examples of planned downtime

Figure 4.2 illustrates a simple schema upgrade proposed for MediaWiki 1.4 that was ultimately rejected because it would have imposed downtime on upgrade. This schema change attempted to improve the performance of the site by replacing the text-based lookups of old article revisions with index queries. After dropping column `old_title` from table `old` ①, some queries issued by the old MediaWiki version would produce SQL errors. Similarly, because the schema upgrade also adds a column `old_id` ②, the new MediaWiki version would be unable to operate on the old schema. To prevent exposing the clients to these errors, the upgrade could be performed in two steps: (i) add column `old_id` and upgrade the wiki engine, in an atomic operation; (ii) drop column `old_title`. During the first step, the clients must not access the system to avoid producing errors.

Moreover, because the values in the new column are copied from another database table ③, this data transfer must mirror the updates enacted by live workload. The new column `old_id` is initialized with the contents of column `cur_id` from a different table, `cur`, selected by joining tables `old` and `cur` on the title column. The offline schema upgrade, proposed for inclusion in MediaWiki, uses three SQL commands (see Figure 4.3a). However, during an online upgrade, the clients can access the system while these queries

```
ALTER TABLE old
  ADD COLUMN old_id
    INT(8) UNSIGNED NOT NULL;

UPDATE old,cur
  SET old_id=cur_id
  WHERE old_title=cur_title;

ALTER TABLE old
  DROP COLUMN old_title;
```

(a) Offline upgrade.

```
INSERT row:
  Into old:     row.old_id ← row ⋈ cur
  Into cur:     old.old_id ← old ⋈ row

UPDATE row.*_title:
  From old:     row.old_id ← row ⋈ cur
  From cur:     old.old_id ← old ⋈ row

UPDATE row.*_id:
  From cur:     old.old_id ← old ⋈ row

DELETE row:
  From old:     do nothing
  From cur:     old.old_id ← default
                WHERE old ⋈ row
```

(b) Online upgrade.

Figure 4.3. Two implementations of the schema upgrade from Figure 4.2.

are executed. The online-upgrade procedure must take into account the effect of the `INSERT/UPDATE/DELETE` queries issued by the live workload and must synchronize the data in the new column, `old.old_id`. This requires reevaluating the join condition `old_title=cur_title` for each live query to determine if the value of `old_id` must be updated—in the row changed by the query or in some other row of table `old` (see Figure 4.3b). Successful `INSERT` queries, in either one of tables `old` and `cur`, change the set of tuples that join and require scanning the other table as well. `UPDATE`s of column `*_title` change the tuples that join, while `UPDATE`s of `cur.cur_id` change data that might need to be copied into `old`. These operations also require scanning the other table, which was not involved in the `UPDATE` query. `DELETE`s of `cur` rows change the tuples that join and require changing some `old_id` instances to the default value because the corresponding rows do not join with `cur` anymore. In general, the stream of queries issued by the live request is not sufficient for synchronizing schema changes that compute the `JOIN` of several tables [Gupta and Mumick, 1995].

Figure 4.4 illustrates the most complex schema change in the upgrade history of Wikipedia, introduced with the MediaWiki version 1.5. Prior to this version, the `cur` table stored the content and metadata of the current revisions of Wikipedia articles, and table `old` stored the previous revisions. The upgrade moved the article-specific metadata into

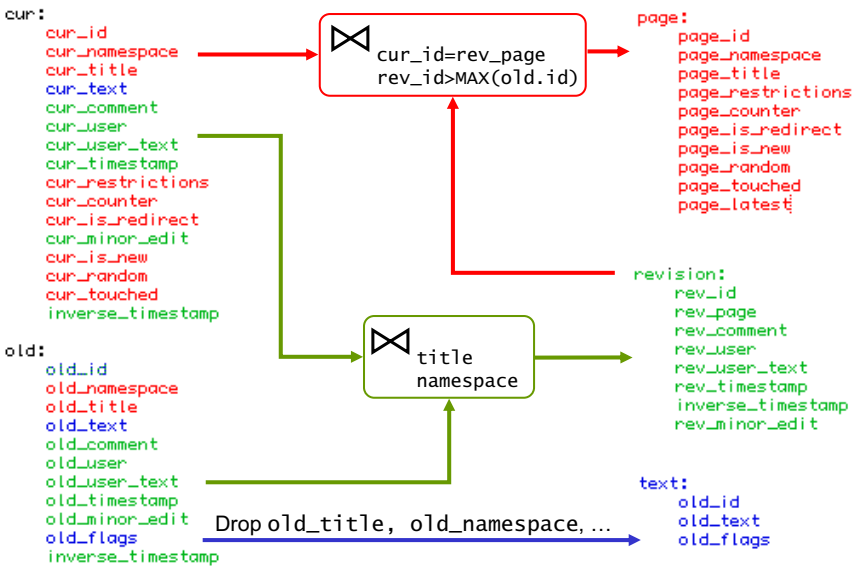


Figure 4.4. Major database-schema reorganization at Wikipedia.

the `page` table, the revision-specific metadata into the `revision` table and the content of the revisions into the `text` table. The goals of this major restructuring were to improve performance (*e.g.*, by separating metadata from content to allow faster aggregation) and to support new features (*e.g.* renaming articles without having to modify all their past revisions). This change was implemented by five developers over a period of one year. During the upgrade, Wikipedia was locked for editing, and the schema was converted to the new version in about 22 hours [Wikimedia Foundation, 2005].

These examples illustrate that synchronizing multiple versions of the database schema during an online upgrade would require a significant development effort. Such online upgrades are error-prone due to their complexity and they can impose a high run-time overhead on the production system.

4.2 Leading causes of planned downtime

The study of Wikipedia provides insight into the causes of planned downtime. The potential or recorded cases of upgrade-induced downtime, such as the ones described in Section 4.1.2, point to modifications of on-disk data structures (*e.g.*, the historical evolution of database schemata) as the main reason for performing upgrades offline, during windows of planned downtime.

4.2.1 Changes to database schemata and data formats

Curino et al. [2008b] show that the 171 evolution steps of MediaWiki's database schema can be modeled using 11 schema-modification operators (SMOs). These operators specify how the upgrade converts the database to a new schema. The first column in Table 4.1 summarizes the fraction of MediaWiki's schema changes that correspond to each SMO.

Four SMOs, `DROP TABLE`, `RENAME TABLE`, `MERGE TABLE`, `RENAME COLUMN`, create new schemas that prevent restarting the MySQL replication during a rolling upgrade, and, therefore, impose downtime. Five additional SMOs cannot be supported during a rolling upgrade because (i) the old application would be unable to query the new schema (`DROP COLUMN`, `MOVE COLUMN`); (ii) `UPDATE` queries would attempt to access rows that do not exist anymore (`DISTRIBUTE TABLE`); or (iii) data dependencies would be broken because changes would be applied only to the source column (`COPY COLUMN`, `MOVE COLUMN`). The most common operator, `ADD COLUMN`, is usually compatible with rolling upgrades because it inserts constant values into the new column. However, in a few cases, `ADD COLUMN` adds data dependencies, by inserting values based on other columns from the same table, or creates columns with values incremented automatically, which might not produce the same ordering on the master and the slave.

In a sequence of schema modifications (*e.g.*, to describe a MediaWiki upgrade from V1.4 to V1.5), a SMO can cancel the effects of a previous one; for instance, if `CREATE TABLE X` precedes `DROP TABLE X`, these changes do not impose downtime during the upgrade. This suggests that individual evolution steps are not sufficient for determining whether downtime will be required. For instance, while Wikipedia always uses the most recent MediaWiki version and has deployed all the past versions of the wiki engine sequentially, in practice software upgrades often skip versions. I consider the SMO sequences that define all the possible upgrades among MediaWiki versions V1.1 – V1.11 (in this chapter, I do not consider downgrades, which would require the inverse operations).

Figure 4.5 shows the likely outcome of these upgrades. 38 out of the 55 upgrades would introduce changes that prevent restarting the MySQL replication, and in 12 additional cases the changes would prevent a rolling upgrade. These upgrades require planned downtime. Only 5 MediaWiki upgrades can be performed online, through a rolling upgrade. These upgrades typically require upgrading the site to the subsequent version of

Table 4.1. Database schema changes in Wikipedia.

Schema change (%)	Description	Rep	RU
`CREATE TABLE` (4.9%)	Creates new, empty table.	Y	Y
`DROP TABLE` (1.8%)	Remove existing table.	N	N
`RENAME TABLE` (0.6%)	Rename table, without affecting the data.	N	N
`DISTRIBUTE TABLE` (0%)	Distribute rows of a source table into two new tables, according to a condition.	Y	N
`MERGE TABLE` (0.8%)	Create a new table as the union of two tables with the same schema.	N	N
`COPY TABLE` (1.2%)	Duplicate existing table.	Y	Y
`ADD COLUMN` (21.2%)	Add column and populate it with values generated by a constant or a function.	Y	Y/N
`DROP COLUMN` (14.5%)	Remove existing column.	Y	N
`RENAME COLUMN` (8.8%)	Change column name, without affecting the data.	N	N
`COPY COLUMN` (0.8%)	Copy column into another table, according to a join condition.	Y	N
`MOVE COLUMN` (0.2%)	`COPY COLUMN`, then drop the original.	Y	N

Rep = Supported by MySQL replication (Yes/No).

RU = Supported during a rolling upgrade (Yes/No).

MediaWiki. However, the major changes introduced in versions 1.5 and 1.7 are incompatible with the database replication. Additionally, versions 1.4 and 1.10 introduced auto-incremented columns, version 1.3 dropped a column, and version 1.6 added a column with dependencies on other columns.

Upgrading commercial systems can require even more complex schema changes. Examples of changes from Oracle's enterprise systems include accessing multiple rows in the source table (through a self-join), creating a primary key from a column that allows repeated values, and initializing new columns with aggregate values, which are difficult to maintain incrementally (*e.g.* computing `MAX(`*column*`)` when values from *column* may be deleted by the live workload) [Downing, 2008]. Moreover, the aggregates sometimes represent only placeholder values, rather than invariants of the new schema, which suggests that the upgrade semantics can not be determined, in all the cases, from the specification of the schema evolution.

These challenges are not limited to databases, but they also affect upgrades that modify other persistent data-structures, such as the metadata used by distributed file systems. For

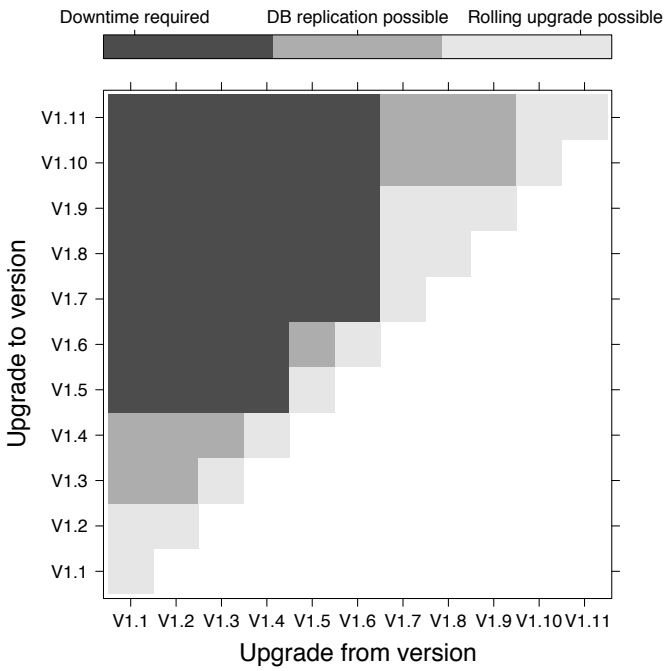


Figure 4.5. Planned downtime imposed by MediaWiki upgrades.

example, new versions of the GPFS parallel file system [Schmuck and Haskin, 2002] are usually deployed without downtime, through rolling upgrades. However, when GPFS's inode structure was updated to allow disk-sector numbers represented on 64 bits (instead of 32 bits only), the upgrade required unmounting the file system for changing the metadata on disk [Schmuck, 2010].

4.2.2 Data conversions

While schema changes represent the leading cause of planned downtime, they are not the only cause. In some cases, the data instances must be converted as well. In MediaWiki's version history, one upgrade has required converting the text from all the current and past article revisions recorded in the database. Starting from version 1.5, MediaWiki supports only the UTF-8 character set, and older wikis using Latin-1 were required to convert their data to the new encoding. This is a long-running operation, which competes with the live workload for querying and modifying the database and which can impose a significant performance overhead. Because of uncertainty about the overhead introduced in the production system, system administrators often prefer to perform such data conversions during offline upgrades [Williams, 2009].

4.2.3 Competitive upgrades

Sometimes, instead of switching to a newer version, the upgrade aims to replace the existing system with a completely different one, which provides similar or equivalent functionality. These upgrades occur when an enterprise changes vendors, and they usually impose downtime because of incompatibilities between the two systems. Wikipedia has performed two such competitive upgrades: (i) when it switched from UseModWiki to a custom-built wiki engine, now remembered only as "Phase II," and (ii) when this code base was rewritten and became MediaWiki.

4.2.4 Changes that do not require downtime

Modifications to database indices, implemented for performance-tuning purposes, are the most frequent type of schema evolution in MediaWiki, accounting for 40.3% of such schema changes [Curino et al., 2008b]. In MySQL, index changes are inefficient and require locking a table in order to rebuild its index. These common schema modifications can impose downtime for systems relying on a single database node.

Past research has dedicated considerable attention to updating indices incrementally, and some commercial database servers support online index-definition [for example: Oracle Corporation, 2009; Microsoft Corporation, 2010]. However, even without this technology, index changes do not impose planned downtime in distributed systems, which employ database replication or clustering. In Wikipedia, for example, such changes are performed online, through a rolling-upgrade, because they do not require application modifications. Similarly, rolling upgrades enable schema changes that implement simple data-type conversions (12.8% of schema changes, in Wikipedia), such as increasing the size of a numeric type (*e.g.*, `INT(8)` $\mapsto$ `INT(16)`).

4.3 Existing techniques for avoiding planned downtime

Distributed enterprise systems often employ *rolling upgrades*, as described in Section 4.1.1, to avoid planned downtime. However, rolling upgrades create mixed database versions, which must interact with the live workload in a consistent manner. For this reason, rolling upgrades are feasible only in the presence of simple schema changes, which are tolerated by the database replication mechanism and which do not introduce data dependencies that

must be synchronized on-the-fly, in response to the live workload (see Section 4.1.2). In general, rolling upgrades rely on the fundamental assumption that the changes implemented are backward-compatible [Brewer, 2001].

Asynchronous database replication [for example: Wong et al., 2009; Quest Software, Inc., 2010] is sometimes used for minimizing downtime during complex upgrades [for example, at Priceline.com: Baltazar, 2005]. Unlike master/slave replication, this mechanism captures the stream of transactions committed in the production database, applies data transformations and replays the changes at another database replica. Because asynchronous replication does not aim to provide strong consistency [Narasimhan, 1999] among replicas, it enables more complex schema and data conversions during the upgrade. However, this approach cannot handle many of the schema changes that have caused downtime for Wikipedia. For example, because the transformations use only the stream of live-workload transactions as input, they cannot implement schema changes that join multiple tables, such as the examples from Section 4.1.2.

Oracle 11g Release 2 [Oracle Corporation, 2009] provides database support for mixed versions, by introducing *edition-based redefinition* [Choi, 2009]. This technique creates two separate views (called editions) over the base tables in the database. The old version of the business logic uses the old edition and the new version uses the new edition, which can have a different schema. When data changes in either edition, cross-edition triggers (backward and forward) ensure that changes are propagated between the two views. Eventually, the old edition is retired. Edition-based redefinition is a technique for on-the-fly data synchronization, but it does not provide support for defining correct data transformations. For example, the invertibility and the correctness of cross-edition triggers are concerns left entirely up to the application programmers. This technique is currently targeted at allowing system administrators to test a new version before deploying it widely [Choi, 2009].

The limitations of data transformations that can be performed online have been studied in the context of *materialized views*, which are defined in terms of one or several base tables and which store the derived tuples in the database. A materialized view must be maintained at runtime, in response to updates to the base tables [Gupta and Mumick, 1995]. While this technique has been used primarily for caching or for data warehousing, the throughly-studied problem of materialized-view maintenance provides insight into the schema changes that can be implemented during an online upgrade. There are many sit-

uations where materialized views cannot be updated on-the-fly, using only the stream of queries from the live workload: when the view definition joins two or more base tables and the workload includes `INSERT` queries, when the view does not have a primary key, when the view or the base tables include triggers (which could cause deadlocks through recursion) or in the presence of integrity constraints. Such practical considerations impose a rescan of the base tables (in database terminology, these views are not self-maintainable), and, similarly, they can prevent online upgrades in the presence of complex schema changes.

Interestingly, some Internet systems can rely on the characteristics of their workload and on the structure of their persistent data to avoid these problems. At Facebook changes to database schemas are usually limited to adding columns and tables, and schema inconsistencies between the application and the database do not constitute a significant challenge[3] [Reiss, 2009]. This is the result of the site's highly-connected user base. Because the friendship connections evolve continuously and do not produce stable clusters, the Facebook system scales better through horizontal partitioning (*e.g.*, by splitting users across several databases) than vertical partitioning (*e.g.*, by splitting the names and addresses of users in different database tables). This allows Facebook to avoid the planned downtime required to implement major schema changes.

4.4 Summary of findings

The causes of planned downtime cannot be understood by focusing exclusively on the database or on the business logic. When upgrading a distributed system end-to-end, developers and administrators must carefully consider the interplay of database schema evolution with the code modifications required and with the workload of the system-under-upgrade.

To determine the common reasons for planned downtime, I study the upgrade history of Wikipedia—currently one of the ten most popular websites. I combine data from a rigorous study of Wikipedia's back-end evolution with information from design documents and archived discussions, and I correlate these findings with examples drawn from three commercial systems (Facebook, GPFS and Oracle's enterprise software). The leading cause of planned downtime are changes to database schemata: when the schema undergoes a

[3] In Chapter 9 I describe a different problem that occurs frequently during Facebook upgrades.

major restructuring, the old version of the business logic can no longer query the new version of the schema. This requires upgrading the business logic in the middle tier and the database in the back-end together, in an atomic operation. Such upgrades impose downtime because the schema conversion is a long-running operation. The need to implement schema changes imposes downtime for 50 out of the 55 possible upgrades among the first eleven versions of MediaWiki (the business logic of Wikipedia). Similar problems occur when changing the format of other persistent structures, such ash the on-disk inodes of a distributed file system.

Other causes of downtime include computationally-intensive data conversions and competitive upgrades. Some upgrades do not stop at changing the database schema, but modify the data itself (*e.g.* by converting the entire content of Wikipedia to the UTF-8 character set). These long-running data conversions compete with the live workload and might overload the database. Competitive upgrades (*i.e.*, replacing the business logic with alternative software that provides similar, but not equivalent functionality) require complex conversions and typically impose downtime.

Our revels now are ended. These our actors, [...]
Are melted into air, into thin air:
And like the baseless fabric of this vision, [...]
We are such stuff as dreams are made on;
And our little life is rounded with a sleep.

William Shakespeare, The Tempest, 1611

Chapter 5

The AIR Properties

THE upgrade properties that have been proposed in the prior work primarily focus on facilitating the development of program updates, on avoiding some of the common runtime errors that could be caused by an upgrade, or on reasoning about the consistency of the persistent data in the system-under-upgrade. I submit that a dependable, online-upgrade mechanism for distributed systems should provide three properties:

- ATOMICITY: At any time, the clients of the system-under-upgrade must access the full functionality of either the old or the new versions, but not both. The end-to-end upgrade must be an atomic operation.
- ISOLATION: The upgrade operations must not change, remove, or affect in any way the dependencies of the old version (including its performance, configuration settings and ability to access the data objects).
- RUNTIME-TESTING: The upgrade mechanism must allow testing the upgraded system under operational conditions.

The ISOLATION property provides an alternative to tracking dependencies. By accessing the old version in a non-intrusive, read-only manner, this approach prevents breaking hidden dependencies during the upgrade. The ATOMICITY and RUNTIME-TESTING properties imply that the system must not include mixed, interacting versions, which synchronize their states in the back-end and that exhibit runtime-emerging behaviors that are difficult to validate through offline testing. The lack of mixed versions enables long-running data conversions in the background, during an online upgrade; a degraded functionality is necessary only during the atomic switchover to the new version. Moreover, because it does not require correctness constraints for the inter-version interactions or knowledge of the old version's

dependencies, this approach reduces the manual interventions needed for preparing the upgrade and is easier to use than the current techniques.

These benefits come at a cost. For instance, in many cases, guaranteeing the ISOLATION property would require additional hardware and storage resources. While this design choice might be inadequate for many resource-constrained systems, the potential cost of downtime in current distributed systems offsets the costs of new hardware or of leasing resources from a public cloud-computing infrastructure [Patterson, 2002; Zolti, 2006; Reiss, 2009; Choi, 2009; Google Inc., 2010]. The next chapter describes the design of a system, called Imago, that provides the AIR properties, and Chapter 7 discusses the impact of these properties on upgrade dependability.

The insect then casts off the spoils of it's former state, and appears in it's *imago* or perfect form.

George Adams, Jr., Essays on the Microscope, 1787

Chapter 6

Design and Implementation of Imago

THE AIR properties, defined in Chapter 5, represent the blueprint for addressing the leading causes of planned and unplanned downtime that results from software upgrades. This chapter presents a system called Imago,[1] which executes end-to-end upgrades in distributed systems and which realizes the ATOMICITY, ISOLATION and RUNTIME-TESTING properties in practice. Imago reduces planned downtime, by performing an online upgrade even in the presence of complex data and schema changes, and avoids upgrade failures due to broken dependencies, by presenting an alternative to dependency-tracking techniques.

The key idea behind Imago is to install the new version in a *parallel universe*—a logically distinct collection of resources, including CPUs, disks, network links, *etc.*—that isolates the production system from the upgrade operations. The new version may be a more recent version of the old production system, or it may be a completely different system that provides similar or equivalent functionality. Imago updates the persistent data of the new version through an opportunistic data-transfer mechanism. The logical isolation between the universe of the old version, $\mathbf{U_{old}}$, and the universe of the new version, $\mathbf{U_{new}}$, ensures that the two parallel universes do not share resources and reduces the impact of procedural errors, which represent the leading cause of upgrade failures (see Chapter 3).

The proof-of-concept implementation described in this chapter provides ISOLATION by using separate hardware resources, but similar isolation properties could be achieved through virtualization. The upgrade process, operating on $\mathbf{U_{new}}$, can not alter the functional dependencies encapsulated in $\mathbf{U_{old}}$. Imago also accommodates mechanisms for min-

[1] The *imago* is the final stage of an insect or animal that undergoes a metamorphosis, *e.g.*, a butterfly after emerging from the chrysalis [Oxford English Dictionary, 1989].

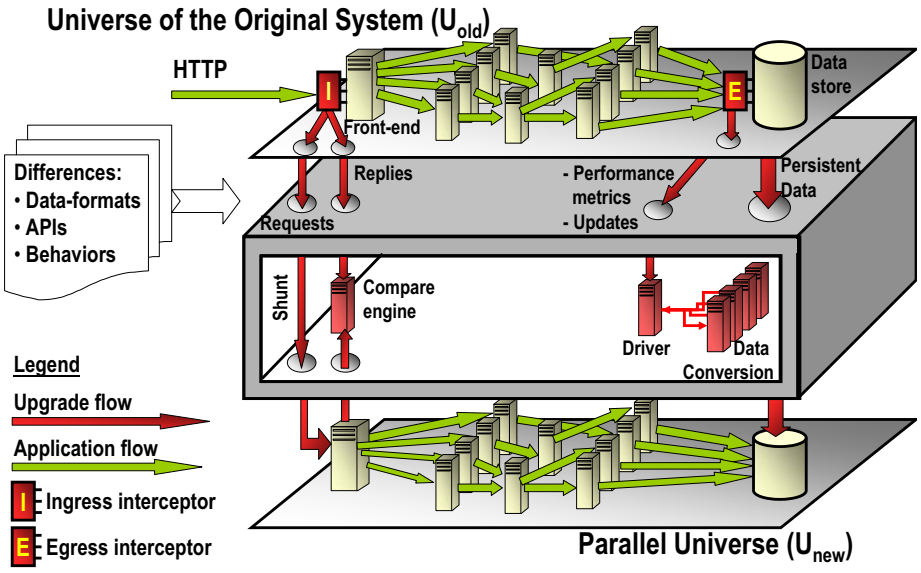


Figure 6.1. Dependable software upgrades with Imago. The old and new versions of a distributed system are installed in parallel universes $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$. Imago intercepts the flow of live requests at the ingress points (**I**) and the egress points (**E**) of the old version and treats the rest of $\mathbf{U_{old}}$ as a black box. The end-to-end upgrade is an atomic operation.

imizing the impact of the upgrade on the live workload, which reduces the risk of breaking non-functional dependencies in $\mathbf{U_{old}}$.

Figure 6.1 illustrates the high-level architecture of Imago. Distributed enterprise-systems typically have one or more *ingress points* (**I**), where clients direct their requests, and one or more *egress points* (**E**), where the persistent data is stored (see Fig. 6.1). The remainder of the infrastructure (*i.e.*, the request paths between **I** and **E**) implements the business-logic and maintains volatile data, such as user sessions or cached data objects. Imago uses an upgrade procedure with five phases: bootstrapping, data-transfer, termination, testing, and switchover [Dumitraș et al., 2007a]. Imago opportunistically transfers the persistent data from the system in $\mathbf{U_{old}}$ to the system in $\mathbf{U_{new}}$, converts it into the new format, monitors the data-updates reaching $\mathbf{U_{old}}$'s egress points and identifies the data objects that need to be re-transferred in order to prevent data staleness. The live workload of the system-under-upgrade, which accesses $\mathbf{U_{old}}$'s data store concurrently with the data-transfer process, can continue to update the persistent data. The egress interceptor, **E**, monitors $\mathbf{U_{old}}$'s data-store activity to ensure that all of the updated or new data objects are eventually (re)-transferred to $\mathbf{U_{new}}$.

Because Imago always performs read-only accesses on $\mathbf{U_{old}}$, the dependencies of the

old version cannot be broken and need not be known in advance. Moreover, **E** monitors the load and the performance of $\mathbf{U_{old}}$'s data store, allowing Imago to regulate its data-transfer rate in order to avoid interfering with the live workload. These mechanisms represent the two keys for providing upgrade Isolation.

Imago implements a distributed coordination protocol that ensures the atomic switchover to the new version when the data transfer is complete. Until this switchover, the progress of the ongoing upgrade is transparent to the clients; after the switchover, only the new version is available, which allows Imago to provide upgrade Atomicity. Imago also enables the live testing of the new version, thus satisfying the Runtime-testing property. Imago takes a holistic approach and focuses on upgrading distributed systems end-to-end, in the presence of major changes in the business logic and of database schema evolution.

Challenge and Contributions

Imago is designed with the goal of improving the dependability of software upgrades in distributed systems by (i) removing the leading cause of upgrade failures—breaking hidden dependencies (see Chapter 3)—and (ii) providing a solution for the leading cause of planned downtime—performing schema and data conversions in the database (see Chapter 4).

Prior work on upgrading distributed systems online [for example: Bloom, 1983; Kramer and Magee, 1990; Ajmani et al., 2006; see also Section 2.3.3] focuses on applications built on top of distributed-object middleware or component frameworks, which allows the framework to provide an online-upgrade mechanisms to all the components of the system. However, real-world enterprise systems are not based on a single, homogeneous framework. Instead, they typically utilize three-tier architectures, with front-end servers that represent the ingress points for client requests, middle-tier servers that implement the system's business logic and back-end data stores that represent the egress points for persistent data. An end-to-end upgrade replaces the old version of the business logic and data schema, used in the production system, with a new version. Such an upgrade requires the coordinated replacement of multiple system components and the conversion of the persistent data to the new format. Volatile data must be converted as well in order to ensure the client-transparency of the upgrade; however, transparency is not always possible for major upgrades that change the interfaces or the semantics of the system. While transparency is not always a goal for such upgrades, in most cases it is desirable to reduce the downtime

by ensuring that either the old or the new version is online, or by providing a gracefully-degraded functionality.

The AIR properties of dependable software upgrades, defined in Chapter 5, are derived from empirical observations and focus on the leading causes of planned and unplanned downtime resulting from distributed-system upgrades. However, the AIR properties are difficult to provide in a practical system. For example, current approaches for online upgrade in distributed systems [*e.g.*, rolling upgrades (see Section2.3.4); Bloom, 1983; Ajmani et al., 2006] focus either on performing upgrades in-place by replacing an existing system—which violates Isolation—or on supporting mixed, interacting versions during the upgrade—which violates Atomicity.

I make a *fundamental trade-off*: in order to implement the AIR properties, Imago imposes a higher resource overhead than previous techniques. This trade-off is based on the observation that, for current distributed systems, the cost of downtime is usually higher than the cost of purchasing or leasing hardware resources [Dumitraș and Narasimhan, 2009a]. Recently, Bhattacharya and Neamtiu [2010] proposed a different trade-off, which involves temporal, rather than spatial, overhead for providing upgrade Atomicity: keeping track of the safe update points on the hosts of the distributed system, and delaying the upgrade until all the hosts are ready to apply it simultaneously.

Assumptions. The design of Imago makes three assumptions. I assume that (1) the system-under-upgrade has well-defined, static ingress and egress points; this assumption simplifies the task of monitoring the request-flow through $\mathbf{U_{old}}$ and the switchover to $\mathbf{U_{new}}$. I further assume that (2) the workload is dominated by read-only requests; this assumption is needed for guaranteeing the eventual termination of the opportunistic data-transfer. Finally, I assume that the system-under-upgrade provides hooks for: (3a) flushing in-progress updates to the persistent data store (needed before switchover); and (3b) reading from $\mathbf{U_{old}}$'s data-store without locking objects or obstructing the live requests in any other way (to avoid interfering with the live workload). I do not assume any knowledge of the internal communication paths between the ingress and egress points.

These assumptions define the class of distributed systems that can be upgraded using Imago. For example, enterprise systems with three-tier architectures satisfy these assumptions. An ingress point typically corresponds to a front-end proxy or a load balancer, and an egress point corresponds to a master database in the back-end. E-commerce web sites

usually have read-mostly workloads [Menascé, 2002], satisfying the second assumption. The two $\mathbf{U_{old}}$ hooks required in the third assumption are also common in enterprise systems: most application servers flush the in-progress updates to their associated persistent storage before shutdown, and most modern databases support snapshot isolation[2] as an alternative to locking.

Non-goals. Imago does not focus on disseminating minor updates, such as fine-grained bug fixes or security patches. Furthermore, Imago does aim to eliminate upgrade failures due to software defects or to upgrade distributed systems in a fully-transparent manner.

Imago takes a holistic approach, focusing on dependable, end-to-end upgrades of distributed systems. The novel characteristics that facilitate this goal are:

- Imago avoids breaking hidden dependencies—the leading cause of upgrade failures—by presenting an alternative to the current approaches based on dependency tracking (see Section 2.3.1). While relying on the knowledge of the planned changes in data-formats and observable system behavior, Imago treats the system-under-upgrade as a black box. Imago provides Isolation by installing the new version in a parallel universe and by transferring the persistent data, opportunistically, into the new version. Imago accesses the universe of the old version in a read-only manner, isolating the production system from the upgrade operations.
- Imago also enables the integration of database schema changes and of long-running data conversions in an online upgrade. This mechanism provides a solution for the leading causes of planned downtime related to software upgrades.
- When the data transfer is complete, Imago performs the switchover to the new version, completing the end-to-end upgrade as an atomic operation. This mechanism allows Imago to provide Atomicity.
- In addition to the traditional testing approaches, Imago provides an opportunity for testing the new version under operational conditions, using the live workload, without exposing the systems's clients to the potential errors that restult from failed tests. This allows Imago to provide Runtime-testing.

[2]This mechanism relies on the multi-versioning of database tables to query a snapshot of the database that only reflects committed transactions and is not involved in subsequent updates [Berenson et al., 1995].

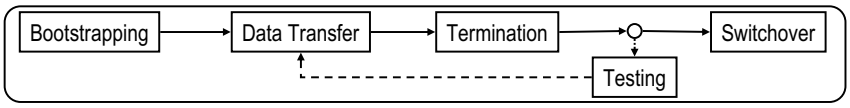


Figure 6.2. Imago's upgrade procedure.

- Through the AIR properties, Imago enables an upgrade-as-a-service approach, which reduces the effort required for implementing software upgrade on a large scale.

Section 6.1 describes the upgrade procedure and the high-level design of Imago. Section 6.2 presents the implementation details. Section 6.3 discusses how the end-to-end mechanisms incorporated in Imago can be used to provide upgrade-as-a-service.

6.1 AIR upgrades with Imago

Imago upgrades distributed systems using a procedure with five phases: bootstrapping, data-transfer, termination, testing, and switchover. Figure 6.2 illustrates the sequence of these upgrade phases.

Bootstrapping. The upgrade starts by installing and configuring the new version in the parallel universe $\mathbf{U_{new}}$. Interceptors **I** and **E** are attached to the ingress and egress points of the production system, which continues to utilize the old version in $\mathbf{U_{old}}$. The data-transfer is initialized by retrieving, from the data store of $\mathbf{U_{old}}$, the list of persistent data-objects to be transferred to $\mathbf{U_{new}}$ (see Figure 6.3). At the end of this phase, the new version is functional but has an empty data-store. The ingress and egress interceptors (**I** and **E**) monitor the flow of live requests through the system from $\mathbf{U_{old}}$.

Data Transfer. Imago transfers persistent data from $\mathbf{U_{old}}$ opportunistically, when the data store has the spare capacity needed to process the data-transfer requests. All the features of the old version are available to the clients during the transfer. The live workload, which accesses the data store in $\mathbf{U_{old}}$ concurrently with the data-transfer process, is allowed to modify the persistent data during the upgrade. The egress interceptor, **E**, monitors $\mathbf{U_{old}}$'s data-store activity to ensure that all of the updated or new data objects are eventually (re)-transferred to $\mathbf{U_{new}}$. **E** detects a competing live workload by examining the rate of queries that do not originate from Imago.

PHASE I: **BOOTSTRAPPING**

1 Retrieve the *id*s of the persistent data-items to be transferred from $\mathbf{U_{old}}$
2 $\forall\, id \in \mathbf{U_{old}}$, $\mathbf{UT}[id] \leftarrow <$ VALID, $\neg$TRANSFERRED $>$ ▷ Initialize the upgrade tracker **UT**
3 Initialize the interceptors **I** and **E** ▷ The interceptors continue to
▷ operate until the switchover
4 **if** **E** detects that data item identified by id' is updated
then **UT** $[id'] \leftarrow <$ INVALID, $\cdot >$

PHASE II: **DATA TRANSFER**

1 **while** $\exists\, id \in \mathbf{UT}$ such that $\mathbf{UT}[id] \neq <$ VALID, TRANSFERRED $>$
2 **do** $id \leftarrow \text{top}(\mathbf{UT})$
3 Query id from the data-store of $\mathbf{U_{old}}$
4 Convert id to the data schema of $\mathbf{U_{new}}$
5 Insert id into the data store of $\mathbf{U_{new}}$
6 $\mathbf{UT}[id] \leftarrow <$ VALID, TRANSFERRED $>$

PHASE III: **DATA-TRANSFER TERMINATION**

if **I** detects write request *req* ▷ Enforce quiescence
then reject *req*
1 Flush all caches from $\mathbf{U_{old}}$
2 Finish data transfer (phase II)
3 **if** testing required
4 **then goto** phase IV
5 **else goto** phase V

PHASE IV: **TESTING**

1 Checkpoint $\mathbf{U_{new}}$
2 **while** testing required
3 **do** $\forall$ *req* detected by **I** ▷ Shunt live requests to $\mathbf{U_{new}}$
4 send *req* to $\mathbf{U_{old}}$ and to $\mathbf{U_{new}}$
5 receive *reply1* from $\mathbf{U_{old}}$ and *reply2* from $\mathbf{U_{new}}$
6 send $< req, reply1, reply2 >$ to compare engine ▷ See Figure 6.1
7 Rollback $\mathbf{U_{new}}$ to checkpoint taken in step 1
8 **goto** phase II

PHASE V: **SWITCHOVER**

1 Discard volatile state
2 Send all requests to $\mathbf{U_{new}}$
3 Stop interceptors **I** and **E**

Figure 6.3. Pseudocode of the upgrade procedure used by Imago for enforcing the AIR properties.

During the data-transfer phase, Imago also executes the schema and data conversions required by the upgrade. The schema changes are specified using the SMO operators introduced in Section 4.2.1 and are executed outside of the $\mathbf{U_{old}}$ universe, to avoid overloading the production system. Unlike the existing approaches for upgrading distributed systems, Imago does not upgrade the system in-place. Because the application never interacts with a database schema belonging to a different version, Imago trivially supports the `DROP TABLE`, `RENAME TABLE`, `DROP COLUMN`, and `RENAME COLUMN` schema changes. During the data transfer, the live workload accesses only the schema from $\mathbf{U_{old}}$, which simplifies the implementation of the more complex schema changes.

Figure 6.4 describes how Imago handles the other schema changes that commonly impose planned downtime. For the `DISTRIBUTE TABLE` SMO, Imago evaluates the *condition* during the data transfer and determines whether to apply a change to S or T, while for `MERGE TABLE`, all changes to S or T will be applied to R. For new auto-incremented columns, Imago assigns the new value during the data transfer. For new columns initialized with a function based on other columns from the same table, Imago monitors the updates to the table and applies the updates correctly, to both the source and the destination columns.

`COPY COLUMN` is a more complex transformation because it joins two tables to determine which values are copied into the new column. If the transfer of tables R and S from $\mathbf{U_{old}}$ is ongoing, Imago saves the stream of updates for applying it later; otherwise, Imago applies the update and re-evaluates the set of values that must be copied into the new column. With Imago, the schema changes are not required the table join can be evaluated repeatedly in the parallel universe $\mathbf{U_{new}}$, without the risk of overloading the production system. Imago could also use more sophisticated techniques for deferring the maintenance of join expressions [for example: Salem et al., 2000]. Unlike in the classical materialized-views maintenance problem (see Section 4.3), the $\mathbf{U_{new}}$ data store is not required to be in a consistent state during the upgrade, which allows Imago to defer updates until the end of the data transfer.

Similarly, Imago can perform computationally-intensive data conversions during the online upgrade. The conversions are executed outside of $\mathbf{U_{old}}$, and they do not interfere with the production data store. Imago's mechanisms for incorporating schema and data conversions in an online upgrade can also be extended to provide support for performing competitive upgrades without planned downtime.

Termination. The data transfer will eventually terminate if the transfer rate exceeds

```
DISTRIBUTE TABLE R INTO S with condition, T
1  if condition
2     then Apply change to S
3     else Apply change to T

MERGE TABLE S,T INTO R
1  Apply update to new table R

ADD COLUMN C auto-increment INTO R
1  Assign auto-incremented value during data transfer

ADD COLUMN C FUNC(A) INTO R
1  if A is updated
2     then C ← FUNC(A)

COPY COLUMN C FROM R INTO S WHERE join-cond
1  if transfer of R and S is complete
2     then Compute R ⋈_join-cond S at the destination
3     else Save the update and apply it later
```

Figure 6.4. Performing database-schema changes, without planned downtime, using Imago. The schema changes are specified using the schema modification operators (SMOs) introduced by Curino et al. [2008b].

the rate at which $\mathbf{U_{old}}$'s data is updated (this is easily achieved for read-mostly workloads). To complete the transfer of the remaining in-progress updates, Imago must enforce a brief period of quiescence for $\mathbf{U_{old}}$. This is required before advancing to either the switchover or the testing phases.

Switchover. Switching atomically to the new version is often the most challenging requirement for an online-upgrade mechanism. Imago can enforce quiescence either using the **E** interceptor, by marking all the database tables read-only and and by mounting the filesystems with the `ro` option, or using the **I** interceptors, by blocking all the incoming write requests. The first option is straightforward: the database and the filesystems prevent the live workload from updating the persistent state of $\mathbf{U_{old}}$, allowing the data-transfer to terminate. This approach is commonly used in the industry due to its simplicity [Oracle Corporation, 2008].

If the system-under-upgrade can not tolerate the sudden loss of write-access to the

The upgrade driver executes:

```
    ▷ Join the group of ingress interceptors
1   JOIN (IGrp)
2   Wait until the data-transfer is nearly completed
3   BCAST (flush)
4   while ∃I ∈ IGrp : I has not delivered flush-done
5       do DELIVER (msg)
6          if msg = self-disconnect
7             then JOIN (IGrp)
8          elseif msg ∈ {self-join, interceptor-join}
9             then BCAST (flush)
    ▷ Received flush-done from all live interceptors
10  Complete data-transfer
11  Send all requests to U_new
12  BCAST (shutdown)
```

Each ingress interceptor **I** executes:

```
    ▷ Join the group of ingress interceptors
1   JOIN (IGrp)
2   DELIVER (msg)
3   if msg = flush
4     then Block incoming write requests
5          for ∀ host ∈ {middle-tier connections}
6              do
                  ▷ Flush in-progress requests
7                 FLUSH (host)
8          BCAST (flush-done)
9          while (TRUE)
10             do DELIVER (msg)
11                if msg = self-disconnect
12                   then JOIN (IGrp)
13                elseif msg ∈ {flush, driver-join}
14                   then BCAST (flush-done)
15                elseif msg = shutdown
16                   then Shut down I
```

Figure 6.5. Pseudocode of the atomic switchover protocol, which allows the ingress interceptors to determine when all the in-progress updates have been flushed to the persistent data store of $\mathbf{U_{old}}$.

database, Imago can instruct the **I** interceptors to block all the requests that might update $\mathbf{U_{old}}$'s persistent data (read-only requests are allowed to proceed). In this case, Imago must flush the in-progress requests to $\mathbf{U_{old}}$'s data store in order to complete the transfer to $\mathbf{U_{new}}$. Imago does not monitor the business logic of $\mathbf{U_{old}}$, but the **I** interceptors record the active connections of the corresponding ingress servers to application servers in the middle tier and invoke the flush-hooks of these application servers. When all the interceptors report the completion of the flush operations, the states of the old and new systems are synchronized, and Imago can complete the switchover by redirecting all the traffic to $\mathbf{U_{new}}$ (this protocol is described in Fig. 6.5). The volatile data (*e.g.*, the user sessions) is not transferred to $\mathbf{U_{new}}$ and is reinitialized after switching to the new version. Until this phase the progress of the ongoing upgrade is transparent to the clients, but after the switchover only the new version will be available.

Testing. Imago can also perform a series of iterative testing phases before switching to the new version. Imago checkpoints the state of the system in $\mathbf{U_{new}}$ and then performs

offline testing—using pre-recorded or synthetically-generated traces that check the coverage of all of the expected features and behaviors—and *online testing*—using the live requests recorded at **I**. In the second case, the testing environment is nearly identical to the environment that becomes the production system after the switchover,[3] which ensures that the testing results are representative for the behavior of the upgraded system. Quiescence is not enforced during the testing phase, and the system in $\mathbf{U_{old}}$ resumes normal operation while **E** continues to monitor the persistent-state updates. At the end of this phase, Imago rolls the state of the system in $\mathbf{U_{new}}$ back to the previous checkpoint, and the data transfer resumes in order to account for any updates that might have been missed while testing.

After adequate testing, the upgrade can be rolled back, by simply discarding the $\mathbf{U_{new}}$ universe, or committed, by making $\mathbf{U_{new}}$ the production system. Because Imago does not rely on any knowledge of the internal communication paths between the ingress and egress points of $\mathbf{U_{old}}$ and because all of the changes required by the upgrade are made into $\mathbf{U_{new}}$, the ISOLATION property is guaranteed and the upgrade does not break any hidden dependencies in $\mathbf{U_{old}}$. Similarly, the atomic switchover protocol allows Imago to provide ATOMICITY, and the testing phases enable RUNTIME-TESTING.

6.2 Implementation details

Imago has four components (see Fig. 6.1): the *upgrade driver*, which transfers data items from the data store of $\mathbf{U_{old}}$ to that of $\mathbf{U_{new}}$ and coordinates the upgrade procedure, the *compare-engine*, which checks the outputs of $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$ during the testing phase, and the **I** and **E** interceptors. The upgrade driver and the compare engine are processes that execute on hardware located outside of the $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$ universes, while **I** and **E** are associated with the ingress and egress points of $\mathbf{U_{old}}$.

6.2.1 Upgrade driver

Imago uses a data structure, called the *upgrade tracker* (**UT**), to remember the persistent data objects from $\mathbf{U_{old}}$ that have already been transferred to $\mathbf{U_{new}}$. In addition to the information required to identify each data object, the upgrade tracker also stores the status of

[3] There are two exceptions: $\mathbf{U_{new}}$ receives the live client requests through a shunt at **I**, and the new version's replies are not propagated to the clients.

the item during the data-transfer phase: TRANSFERRED (the item has been transferred and is identical in both $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$), ¬TRANSFERRED (the item has not yet been transferred from $\mathbf{U_{old}}$) or UPDATED (the item was updated in $\mathbf{U_{old}}$ since its last transfer). When Imago, using information received from **E**, detects that a data item has been updated or inserted in the $\mathbf{U_{old}}$ database, it updates the status of the the corresponding entry in the upgrade tracker and then it (re)-schedules the item for a fresh transfer to $\mathbf{U_{new}}$ (see Figure 6.3, line 4). The upgrade tracker also maintains the order in which data items are scheduled for transfer. This feature can be used to prioritize the transfer of certain data items. Extracting the item with the highest transfer-priority from the upgrade tracker has a constant complexity, $O(1)$.

Adaptation mechanisms. The **E** interceptor also reports system-level statistics, such as the CPU load, the memory usage and the number of page faults, using the `/proc` interface. To avoid overloading the database server, **E** passively records these statistics and forwards them to the upgrade driver without performing any processing. The upgrade driver assesses the load of the database machine and the query-arrival rate, by filtering its own queries out of the information provided by **E** in order to focus on the live workload. Imago evaluates the load of $\mathbf{U_{old}}$ every 30 seconds which enables autonomic management techniques that reduce the data-transfer rate when Imago risks overloading $\mathbf{U_{old}}$'s data store. To avoid competing for resources with the live workload, Imago currently uses a simple adaptation policy, which pauses the data transfer when the transfer rate exceeds 5 queries/s.

6.2.2 Egress interceptor (E)

The **E** interceptor performs two functions: (i) recording the live updates to $\mathbf{U_{old}}$'s persistent data objects and (ii) monitoring the database server's performance indicators. I implement the **E** interceptor by transferring the query log of the database to the upgrade driver and by employing a *zero-copy optimization*, which relies on direct memory access (DMA) to avoid copying the content of the file into user space.[4] The log records the queries observed at the egress point of $\mathbf{U_{old}}$ (see Figure 6.6). By parsing the log file entries on-the-fly, the upgrade driver extracts the `INSERT/UPDATE/DELETE` queries from the log and determines which data objects must be re-transferred. This information is sufficient for updating the upgrade

[4] In Imago, this is achieved using the UNIX system call `sendfile()`, which is widely used for optimizing the performance of web servers. In Windows, the analogous system call is `TransmitFile()`.

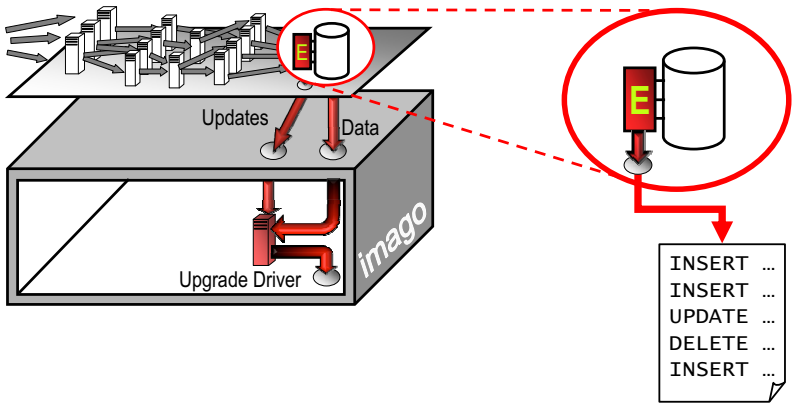


Figure 6.6. Implementation of the egress interceptor.

tracker; Imago does not need to know the timing or the content of the replies to the live queries.

This represents a passive interception technique, which does not block queries for $\mathbf{U_{old}}$'s database server and which imposes no measurable performance penalty on the live workload. However, this implementation suffers from shortcomings that limit its applicability to simple applications that do not use advanced database features. For example, when using database transactions, encountering `COMMIT` in the query log is not sufficient for determining whether the database will commit or abort the transaction and when the new values will become available for transfer. Similarly, the query log does not indicate the values that the database assigns to auto-incremented sequences or which data objects are updated by multi-row queries, which use a range condition in the `WHERE` clause of the `UPDATE` query.

These shortcomings stem from the fact that the workload is currently intercepted before the queries are scheduled and executed by the database management system. Instead, the **E** interceptor could be implemented using the GORDA API [Carvalho et al., 2007], which provides a uniform reflective interface for database processing.[5] The GORDA API allows monitoring the data updates performed by the live workload by retrieving the object-sets from the executor stage of $\mathbf{U_{old}}$'s database, but only after the database considers that the transaction has committed.

[5]As a proof of concept, this reflective API has been implemented, in different ways, for three database servers: PostgreSQL (using a special-purpose trigger library), MySQL (through the C-JDBC clustering middleware [Cecchet et al., 2004]) and Apache Derby (by leveraging the reflexion mechanisms of the Java programming language).

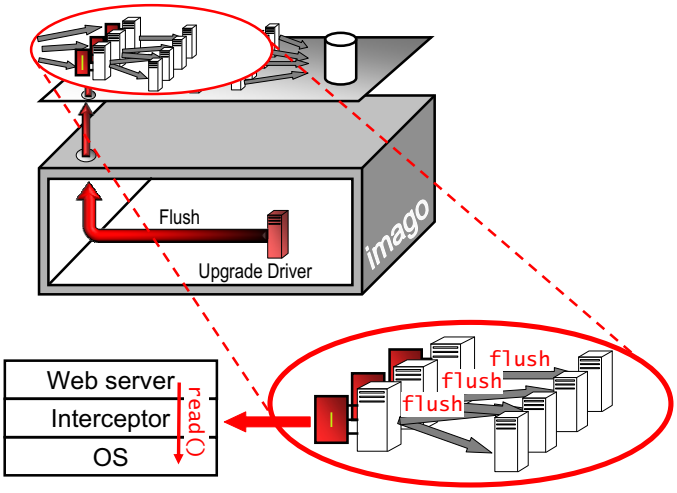


Figure 6.7. Implementation of the ingress interceptor.

Both the implementation based on query-log analysis and the one based on database reflection allow the upgrade driver to determine the current rate of queries received by the database. This information contributes to the second purpose of the **E** interceptor and enables the implementation of autonomic management techniques (see Section 6.2.1).

6.2.3 Ingress interceptor (I)

The **I** interceptor also performs two functions: (i) recording the content of client requests during the testing phase, for copying them into the parallel universe; and (ii) ensuring the atomic switchover to the new version. Unlike for the **E** interceptor, Imago requires an active interception technique at the ingress points, in order to impose quiescence before the switchover. I implement the **I** interceptor using library interposition [Levine, 2000] to redefine system calls used by the front-end web servers. By relying on the dynamic linking capabilities of the linker-loader, these redefined calls are interposed between the application and the system's shared libraries, so that, at runtime, the application transparently calls the interceptor's redefined functions instead of the default ones (see Figure 6.7).

I intercept five system calls: `accept()` and `close()`, which mark the life span of a client connection, `connect()`, which opens a connection to the middle tier, and `read()` and `writev()`, which reveal the content of the requests and replies, respectively. Controlling the behavior of these five system calls is sufficient for implementing the functionality of the **I** interceptor. I maintain a memory pool inside the interceptor, and the rede-

fined `read()` and `writev()` system-calls copy the content of the requests and replies into buffers from this memory pool. The buffers are subsequently processed by separate threads in order to minimize the performance overhead.

In order to complete the data transfer, the upgrade driver invokes the atomic switchover protocol from Figure 6.5. While Imago treats the system-under-upgrade as a black box and does not keep track of the request queuing paths between the **I** and **E** interceptors, the switchover protocol must ensure that no updates to the persistent data in $\mathbf{U_{old}}$ are pending. Imago exploits the fact that the **I** interceptors know—from the observed invocations of the five intercepted system calls—all the active connections to the middle-tier servers. Imago uses this knowledge to flush the in-progress requests before completing the switchover.

The implementation relies on a FLUSH operation, which flushes the in-progress requests from a middle-tier server. Each **I** interceptor invokes the FLUSH operation on the application servers that it has communicated with (see Figure 6.5, line 7). I implement the FLUSH operation for the Apache and JBoss servers. For Apache, I restart the server with the `graceful` switch, allowing the current connections to complete. For JBoss, I change the timestamp of the web-application archive (the `application.war` file), which triggers a redeployment of the application. Both these mechanisms cause the application servers to evict all the relevant data from their caches and to send the in-progress requests to the back-end.

Imago determines the time when all the **I** interceptors are ready to switch through a distributed agreement protocol (see Figure 6.5). I implement this protocol using reliable group-communication primitives: JOIN allows a process to join the group of interceptors and to receive notifications when processes join or disconnect from the group; BCAST reliably sends a message to the entire group; and DELIVER delivers messages in the same order at all the processes in the group. These primitives are provided by the Spread package [Amir et al., 2000]. Imago's switchover protocol provides strong consistency, and it tolerates crashes and restarts of the driver or the interceptors.

6.2.4 Compare engine

The compare engine performs all the computationally-intensive operations required during the testing phase. Each live HTTP session is handled by a separate *compare worker*, which records the input/output traces of $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$. During the testing phase, clients interact only with the system in $\mathbf{U_{old}}$; the system in $\mathbf{U_{new}}$ processes the same requests but without

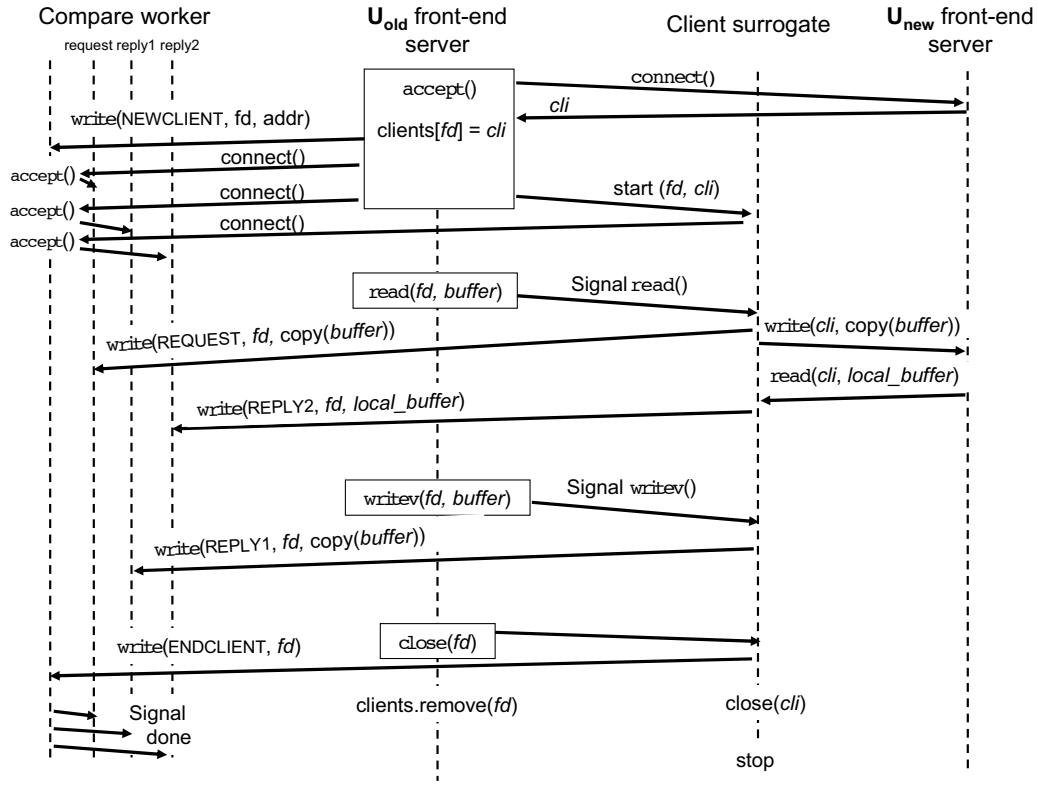


Figure 6.8. Communication protocol used during Imago's testing phase.

sending replies to the clients. The data transfer between $\mathbf{U_{old}}$ and $\mathbf{U_{new}}$ is not active in this phase.

Because $\mathbf{U_{new}}$, does not interact with the live workload, Imago provides *client surrogates* that act as clients for the new version. A client surrogate uses the memory-pool copies of the client requests to forward the same requests to $\mathbf{U_{new}}$, and it receives the corresponding replies (see Figure 6.8). To minimize the overhead, the client surrogates also perform all the blocking I/O operations required for communicating with the compare workers.

Each compare worker maintains a communication channel with an ingress interceptor. The interceptor uses this channel to signal when the corresponding front-end server receives a new client connection. The compare worker records the content of (i) the live requests received from the clients (communicated using REQUEST messages), (ii) the replies generated by the old version, in $\mathbf{U_{old}}$ (communicated using REPLY1 messages) and (iii) the replies generated by the new version, in $\mathbf{U_{new}}$ (communicated using REPLY2 messages).

Table 6.1. Structure of Imago's code.

	Lines of code	Size in memory
Upgrade driver	2,038	216 kB
Egress interceptor	290	
Ingress interceptor	2,056	228 kB
Switchover library	1,464	
Compare engine	571	48 kB
Common libraries	591	44 kB
Application bindings	1,113	108 kB
Total	8,123	—

These three streams of messages enable reasoning about the differences in behavior between the old version and the new version and indicate the performance gains that can be expected from the upgrade.

6.2.5 Structure of Imago's code

All the components of Imago are implemented in C++. Table 6.1 shows the size of the code base and the memory footprints for these components (not including the heap or the stack sizes). The upgrade driver and the egress interceptor are statically linked into a single executable program. The common libraries contain utility algorithms and data structures used by several other components. The application bindings contain all the application-specific routines (*e.g.*, for performing data conversions) and, depending on the system-under-upgrade, constitute between 14%–22% of Imago's code. Some of these application-specific routines would also be necessary for upgrading the system offline, during a window of planned downtime.

6.3 Upgrade-as-a-service

The main disadvantage of Imago is that it requires additional resources, for installing the new version in a parallel universe and for performing the computationally-intensive comparisons and data conversions. However, these additional resources are needed only *during*

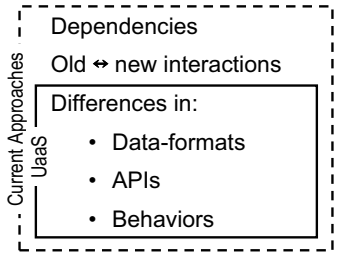


Figure 6.9. Inputs required by the upgrade mechanism.

the upgrade, and they could be leased, temporarily, from an existing cloud-computing infrastructure (*e.g.*, the Amazon Web Services or a distributed system running VMware's vCloud infrastructure). This suggests that Imago enables an *upgrade-as-a-service* (UaaS) model [Dumitraș and Narasimhan, 2009c].

Figure 6.9 illustrates the inputs that developers and system administrators must provide to the upgrade mechanism. Both upgrade-as-a-service and the current approaches for upgrading distributed systems online (see Section 2.3.3) require understanding the differences between the behaviors, interfaces and data formats of the old and new versions. Additionally, the current upgrade mechanisms rely on an in-depth knowledge of the old version's dependencies and require establishing correctness constraints for the interactions between the two versions during the upgrade. In contrast, UaaS cannot break hidden dependencies and does not create states with mixed-interacting versions. This simplicity enables the development of generic upgrade mechanisms, which are applicable to multiple systems and upgrade scenarios. The experience with Imago suggests that at least three quarters of the code required for implementing a complex upgrade can be refactored into reusable upgrade mechanisms, and a much smaller fraction is application specific (see Section 6.2.5).

This illustrates the *separation of concerns* introduced by Imago and UaaS. The functional aspects of the upgrade (*e.g.*, how to convert persistent data to a new format) require application-specific code, and they must be addressed regardless of the upgrade mechanism employed. In contrast, the mechanisms for performing an online upgrade (*e.g.*, live-workload interception, atomic switchover) are applicable to multiple distributed-system upgrades. UaaS allows enterprises to lease all of the hardware resources and most of the software components needed to implement a dependable software upgrade.

Moreover, the current online-upgrade mechanisms limited opportunities for testing the new version and the intermediate steps of the upgrade. By implementing the RUNTIME-

TESTING property, UaaS enables a representative assessment of the new version's behavior. Imago does not force the old and new versions to execute in lock step during the testing phase, which allows it to avoid bypassing the database scheduler in order to control the serialization of concurrent transactions—a technique frequently used in systems for validating data-migrations or for clustering databases [for example: Cecchet et al., 2004; Oliveira et al., 2006; Ding et al., 2010]). Aside from reducing the performance penalty during the upgrade, UaaS allows testing the new version in a realistic environment, where the concurrency-control mechanisms of the database are not disabled. This testing strategy can reveal bugs and misconfigurations in the new version or incompatibilities with the deployment environment.

Upgrade-as-a-service mechanisms harness the opportunities provided by the emerging cloud-computing technologies to simplify large-scale upgrades, to allow upgrades to be executed efficiently online, and to improve their dependability. Upgrades-as-a-service are likely to be more practically usable, less error-prone and better suited to fast upgrade cycles than the current upgrade approaches.

6.4 Summary of findings

Software upgrades performed offline provide the opportunity for conducting extensive tests in order to accepti or reject the outcome of the upgrade. Online upgrades do not have this opportunity; in the presence of mixed versions, system states are often short-lived and cannot be tested adequately, while the system-under-upgrade must recover quickly from any upgrade faults. Unlike the existing strategies for online upgrade, which rely on tracking dependencies, Imago trades off spatial overhead (*i.e.*, additional hardware and storage space) for an increased dependability of the online upgrade.

Imago enforces the AIR properties. A distributed agreement protocol, for atomically switching to the new version, ensures ATOMICITY. A parallel universe, where the upgrade procedure operates without altering the dependencies of the production system, and an opportunistic data-transfer protocol, which ensures that the new version receives all the persistent data converted into the appropriate format, guarantee ISOLATION. An operational-testing mechanism, which allows Imago to use the live workload for testing the new version before the switchover, provides RUNTIME-TESTING.

The focus on upgrading distributed systems from end to end allowed me to rely on certain system properties in order to achieve other goals of the design. For example, Imago is designed to catch up with the live workload eventually, which requires implementing an egress interceptor that monitors the live updates to the persistent data objects. This information also allows Imago to derive the current rate of queries for the production database, which indicates whether the system is under high load, and to regulate its data transfer accordingly. Imago upgrades distributed systems atomically, which requires implementing an ingress interceptor able to enforce quiescence before the final switchover, while the system flushes in-progress requests to the data store in the back-end. This ingress interceptor also enables the operational testing of the new version, using the live workload. Moreover, this testing approach requires recording the content of the user interactions and calls for the use of an active interception technique, such as library interposition. This interception technique also allows Imago to determine, dynamically, which middle-tier servers are actively processing user requests that must be flushed before switching to the new version.

Imago is designed for upgrading enterprise systems with traditional three-tier architectures. The current prototype cannot be readily applied to certain kinds of distributed systems, such as peer-to-peer systems (*e.g.* Chord [Stoica et al., 2001]), which violate Imago's first assumption by accommodating large numbers of dynamically added ingress-points, or data-intensive computing (*e.g.* MapReduce [Dean and Ghemawat, 2004]), which distribute their persistent data throughout the infrastructure and do not have a well-defined egress point. Imago is a proof-of-concept implementation of the AIR properties, which are generic and could be provided, through different mechanisms, to other distributed-system architectures as well.

Imago builds upon the lessons learned from a previous online-upgrade approach, Eternal [Narasimhan, 1999]. Eternal was a fault-tolerant middleware system that orchestrated an atomic upgrade of the old version's replicas, along with all their dependencies. However, some of these dependencies only manifested dynamically, at runtime. In such cases, Eternal required manual assistance (*e.g.* an in-depth pointer analysis), for understanding the nature and depth of the dependency chain. Recent commercial products for rolling upgrades continue to exhibit a similar problem, by requiring the application developers to determine if the interactions among mixed versions are safe [Microsoft Corporation, 2005; Oracle Corporation, 2008].

In contrast, Imago is designed to treat the system-under-upgrade as a black box, and it does not rely on any knowledge of the internal dependencies within the old version. While borrowing a few ingredients (*e.g.* atomic switchover) from Eternal, Imago focuses on implementing *dependency-agnostic upgrades* and employs additional hardware and storage resources. These design choices are based on the intuition—from the past experiences with Eternal from and the current industry trends—that the new hardware needed for implementing the AIR properties costs less than the process of planning an in-place, online upgrade. Enterprises sometimes take advantage of a software upgrade to renew their hardware[6] as well [Zolti, 2006; Downing, 2008]. Moreover, Imago requires additional resources only for implementing and testing the online upgrade. The additional storage and compute cycles can be leased, for the duration of the upgrade, from existing cloud-computing infrastructures (*e.g.* the Amazon Web Services). This suggests that Imago is the first step toward an *upgrade-as-a-service* model. Because it does not require developers and administrators to reason about the potential interactions between the old version and the new version during the upgrade, Imago will likely be easier to use correctly than the current approaches for upgrading distributed systems in-place.

[6]This practice is supported by the fact that the new functionality included in software upgrades usually imposes higher demands on the infrastructure. For example, a Gartner study has found that upgrading SAP R/3 (an enterprise resource planning system) from version 3 to version 4 requires 87% more CPU cycles, 72% more memory and 33% more storage space [Beatty and Williams, 2006].

> How one convinces oneself that a change will operate correctly if it is installed [...] is nontrivial and may be very critical for some applications.
>
> R. Fabry, How to design a system in which modules can be changed on the fly, 1976

Chapter 7

Dependability Benchmarking for Software Upgrades

CHAPTERS 5 and 6 introduced three abstract properties aiming to improve the dependability of software upgrades and described the mechanisms required for implementing these properties in a practical system, but they did not establish the dependability improvements that can be expected from this approach or its limitations. Similarly, the prior evaluations of upgrade mechanisms focus on assessing performance or the range of updates supported, rather than on dependability. This chapter introduces a benchmark for the dependability of distributed-system upgrades, allowing direct comparisons between the downtime and the risk of upgrade failure presented by Imago and by other upgrade approaches.

For example, industry best-practices recommend rolling upgrades, which upgrade-and-reboot one node at a time, in a wave rolling through the cluster. Rolling upgrades cannot perform incompatible upgrades (*e.g.*, changing a component's API). However, this approach is believed to reduce the risks of upgrading because failures are localized and might not affect the entire distributed system [Office of Government Commerce, 2007; Oracle Corporation, 2008].

In this chapter, I challenge this conventional wisdom by showing that AIR upgrades provide more dependability and flexibility. Piecewise, gradual upgrades can cause global system failures by breaking hidden dependencies—dependencies that cannot be detected automatically or that are overlooked because of their complexity. Moreover, completely eliminating software defects would not guarantee the reliability of enterprise upgrades because faults in the upgrade procedure can lead to broken dependencies.

These findings are the result of a novel approach for benchmarking the dependability of

upgrade mechanisms by reproducing the upgrade workflows that are commonly observed to induce downtime. The benchmark focuses on the leading causes of upgrade failures—breaking hidden dependencies—and of planned downtime—changing database schemata—and it enables direct comparisons between the behavior of various upgrade mechanisms in the presence of upgrade faults and of complex schema changes. I use this benchmark to evaluate the dependability of Imago (see Chapter 6) and of two online-upgrade mechanisms frequently used in practice, big flip and rolling upgrades [Brewer, 2001].

Compared with the existing strategies for online upgrades, Imago trades off the need for additional resources for an improved dependability of the online upgrade. While it cannot prevent latent configuration errors, Imago eliminates the internal single-points-of-failure for upgrade faults and the risk of breaking hidden dependencies by overwriting an existing system. Additionally, Imago avoids creating system states with mixed versions, which are difficult to test and to validate. The benchmark results suggest that an atomic, dependency-agnostic approach, such as Imago, can improve the dependability of online software-upgrades in spite of hidden dependencies.

Challenge and Contributions

A large body of anecdotal information suggests that software upgrades are unreliable and often result in downtime or system failures, but this information is incomplete and does not allow a rigorous assessment of the dependability of systems that undergo software upgrades. The known cases of failed upgrades cannot be replicated because we lack information on the system configurations and network topologies used, the detailed upgrade workflows, the errors encountered, *etc.* In consequence, the previous evaluations of upgrade mechanisms focus on performance and runtime overhead, rather than on dependability.

Online software upgrades alter the behavior of distributed systems and interact with their workloads. To enable meaningful comparisons among several upgrade mechanisms, an upgrade-centric benchmark must specify (i) the workflows for the transition from the old version to the new version and (ii) the classes of observable system behavior that correspond to failures. Dependability benchmarking is challenging because, unlike system performance, the dependability attributes (*e.g.*, availability, reliability) cannot be measured

directly. A benchmark that provides quantitative assessments must be based on a large sample of empirical observations, in order to ensure statistical relevance.

The key idea behind the benchmark introduced in this chapter is to rely on the upgrade-centric fault model (Chapter 3) to assess whether an upgrade mechanism is prone to failures and on the types of schema changes that are difficult to integrate in an online upgrade (Chapter 4) to assess whether the same mechanism requires planned downtime. Understanding the leading causes of upgrade failures and the leading causes of planned downtime allows me to conduct *fault-injection experiments* that produce representative results for the dependability of the mechanisms evaluated. Because upgrade mechanisms can make different trade-offs, which might be appropriate for different systems and settings, I avoid reporting a single number indicating which mechanism is the best [DeWitt, 1993]. Instead, the goal of the fault-injection experiments is to determine the qualitative reasons for unavailability during online upgrades in distributed systems, as a first step toward a comprehensive approach for dependability benchmarking.

Assumptions. The outcome of some upgrade faults is difficult to quantify because they have no measurable impact on the system's throughput or response time. For example, a latent error is, by definition, a condition that is benign in the current state of the system but that might be exposed by future changes in the workload or the system configuration. In this chapter, I assume that the effect of a latent error is the same as the effect of a full outage (*i.e.*, the system becomes unavailable). I also assume that an upgrade can be stopped as soon as a problem is identified, instead of allowing the procedure to continue to disable other components of the system-under-upgrade. Moreover, while throughput degradations, response-time increases or failures of database queries can point to infrastructure failures, protocol- and application-level errors are hard to determine through such black-box metrics.[1] Because the semantic of application errors is domain-specific, the upgrade administrator must ultimately rely on the system's error reporting mechanisms to determine whether the upgrade has failed. In this chapter I assume that all errors are detected, so that, even if completing a failed upgrade would have induced a total loss of throughput

[1]For example, when an Internet system starts producing HTTP errors for all the incoming requests, the throughput usually increases because the replies are generated by the front-end servers without involving the rest of the system.

(*e.g.*, by disabling all the servers in the front-end), the outcome is considered to be a partial loss of availability rather than a full outage.

Non-goals. The benchmark does not cover the secondary causes of planned and unplanned downtime, such as competitive upgrades or software defects. The benchmark is also not definitive; like the TPC and SPEC benchmarks, which focus on performance evaluation, the upgrade-dependability benchmark should be updated periodically by reevaluating the leading causes of downtime.

This chapter makes three contributions:

- I propose benchmarking the dependability of distributed systems through a systematic fault-injection approach, using the upgrade-centric fault model introduced in this dissertation. Unlike the prior approaches for dependability benchmarking [Kanoun and Spainhower, 2008], the benchmark introduced in this chapter focuses on failures and downtime that result from software upgrades, rather than from hardware or software defects.
- I propose assessing the planned-downtime requirements expected from an upgrade mechanism through an analysis of the database-schema changes that the mechanism supports without causing unavailability. Unlike the prior approaches for automating database-schema evolution [for example: Curino et al., 2008a], this assessment focuses on the schema changes that prevent online upgrades and impose planned downtime.
- I evaluate the benefits of AIR software upgrades using this benchmarking approach. Imago provides a better availability in the presence of upgrade faults than two alternative approaches, rolling upgrade and big flip (result significant at the $p = 0.01$ level).

Section 7.1 describes the new dependability benchmark. I use this benchmark to assess the availability of systems undergoing software upgrades, in the fault-free case (Section 7.2) and in the presence of upgrade faults (Section 7.3). Section 7.4 focuses on benchmarking the system reliability.

7.1 A benchmark for upgrade dependability

While the upgrade-dependability benchmark is able to produce statistically-significant results indicating which upgrade mechanism provides better availability, the *quantitative* improvements depend on the system architecture and on the specific faults injected, and they might not be reproducible for a different system-under-upgrade. The primary goal of this benchmark is to exhibit the *qualitative* reasons for unavailability during online upgrades, and to emphasize the opportunities for improving the current state of the art in software upgrades.

7.1.1 Upgrade mechanisms and their trade-offs

Dynamic software update (DSU) mechanisms [for example: Segal and Frieder, 1993; Neamtiu et al., 2006; Arnold and Kaashoek, 2009] enable online upgrades for systems that lack hardware redundancy. These mechanisms require programmers to annotate (*e.g.*, by indicating suitable update points in the source code) or to modify the source code of the old and new versions. DSU can introduce new sources of errors (*e.g.*, if the update is applied at the wrong time [Hayden et al., 2009]) and the behavior of a system undergoing dynamic updates is not guaranteed to conform to the specification of either the old or the new version and is difficult to validate in advance [Segal, 2002]. Moreover, active code (*i.e.*, functions on the call stack of the running program) cannot be replaced easily, and updating multi-threaded programs remains a challenging task [Neamtiu and Hicks, 2009].

Industry best-practices recommend carefully planning the upgrades and minimizing their risks by deploying the new version gradually, in successive stages [Office of Government Commerce, 2007]. For example, two widely used upgrading approaches are the *rolling upgrades* and the *big flip* [Brewer, 2001]. The first approach upgrades and then reboots each node, in a wave rolling through the cluster. The second approach upgrades half of the nodes while the other half continues to process requests, and then the two halves are switched. Both these approaches attempt to minimize the downtime by performing an online upgrade. A big flip upgrade trades resources for dependability, by reducing the system's capacity to 50% during the online upgrade in order to isolate the production system from the upgrade operations. However, most systems cannot be divided cleanly in two equivalent halves and some of the resources must be shared by the two halves during the

upgrade. For instance, the persistent data is read and written to by both halves, and an upgrade that modifies the structure or the content of the persistent data might induce a failure in the online half. A rolling upgrade imposes very little capacity loss, but it requires the old and new versions to interact with the data store and with each other in a compatible manner. Ajmani et al. [2006] address some of the limitations of rolling upgrades by enabling arbitrary code modifications, by synchronizing the states of multiple versions and by introducing techniques for ensuring safety at runtime (*e.g.*, temporarily disallowing certain requests).

However, distributed-system upgrades remain vulnerable to upgrade faults of Types 1–4, which break hidden dependencies (see Chapter 3). Existing systems mitigate these problems through two approaches: discovering dependencies automatically and testing the old version and the new version side-by-side, using a sandboxed environment. For example, Galapagos [Magoutis et al., 2008] analyzes the content of configuration files to discover the relationships among storage objects, while Dig et al. [2006] propose detecting source-code refactorings through a combination of syntactic and semantic analyses. These approaches are unable to detect the complete set of dependencies that might be broken during a software upgrade.

Sandbox-testing approaches [for example: Nagaraja et al., 2004; Oliveira et al., 2006; Ding et al., 2010] require knowledge of the systems's behavior, *e.g.*, the communication protocols used and the forms of non-determinism that are allowed. For instance, routing requests to a different application server in the sandbox environment would produce equivalent results, but processing database transactions in a different order would cause the test to fail. To enforce a common order of execution, database requests must be serialized in order to prevent transaction concurrency, for both the production and sandbox databases [Oliveira et al., 2006; Ding et al., 2010]. Aside from inducing a performance penalty during the upgrade, this intrusive technique prevents testing the upgrade's impact on the concurrency-control mechanisms of the database, which limits the usefulness of the validation results. Moreover, some errors remain latent if the components are tested in isolation [Nagaraja et al., 2004].

Compared with these approaches, Imago does not change the way requests are processed in the production system and only requires knowledge of the ingress and egress points. Imago aims to avoid planned downtime by enabling long-running,

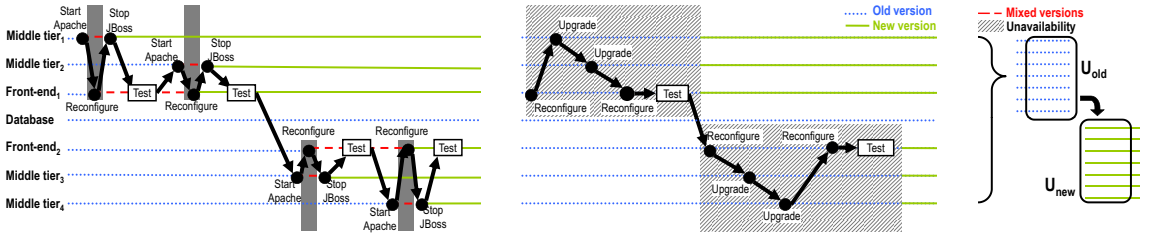
computationally-intensive conversions to a new data format. However, unlike DSU, rolling upgrades and other techniques for in-place upgrade, Imago never produces mixed versions and does not have to establish correctness conditions for the interactions among these versions. Imago is similar to a big flip in that it trades resource overhead for improving dependability and it performs the end-to-end upgrade as an atomic action.

7.1.2 Downtime workflows

Benchmarks that focus on performance evaluations [for example: DeWitt, 1993; Amza et al., 2002; Cooper et al., 2010] specify rules for generating input workloads that are representative for real-world systems. However, software upgrades alter the behavior of the systems being evaluated. A benchmark for upgrade mechanisms should concentrate on the procedure used during the upgrade. A benchmark focusing on dependability should specify upgrade workflows that are representative for the causes of unavailability and failures resulting from software upgrades in real-world systems.

Upgrade scenario. The dependability benchmark uses the Rice University Bidding System (RUBiS) [Amza et al., 2002], an open-source online bidding system modeled after eBay, as the system-under-upgrade. RUBiS has been studied extensively, and several of its misconfiguration- and failure-modes have been previously reported [Nagaraja et al., 2004; Candea et al., 2004; Oliveira et al., 2006; Zheng et al., 2007]. RUBiS has multiple implementations (*e.g.*, using PHP, EJB, Java Servlets) that provide the same functionality and that use the same database schema. I study an upgrade scenario whose goal is to upgrade RUBiS from the version using Enterprise Java Beans (EJB) to the version implemented in PHP. RUBiS is deployed in a three-tier infrastructure, comprising a front-end with two Apache web servers (acting as proxies), a middle tier with four Apache servers that execute the business logic of RUBiS, and a MySQL database in the back-end. More specifically, the upgrade aims to replace the JBoss servers in the middle tier with four Apache servers where we deploy the PHP scripts that implement RUBiS's functionality. The RUBiS database contains 8.5 million data objects, including 1 million items for sale and 5 million bids. I use RUBiS's two standard workloads, based on the TPC-W specification [Menascé, 2002], which are typical for e-commerce web sites.

I conduct experiments in a cluster with 10 machines (Pentium 4 at 2.4 GHz, 512 MB RAM), connected by a 100 Mbps LAN. The performance bottleneck in this system is the



(a) Rolling upgrade. (b) Big-flip upgrade. (c) Imago.

Figure 7.1. Current approaches for online upgrade in distributed enterprise systems. A rolling upgrade modifies, and then reboots, each node at a time, in a wave rolling through the distributed system. A big-flip upgrades one half of the nodes while the other half continue servicing requests. Imago duplicates the entire architecture, transferring all the persistent data to $\mathbf{U_{new}}$.

amount of physical memory in the front-end web servers, which limits the system's capacity to 100 simultaneous clients.

Upgrade procedures. I compare three online-upgrade mechanisms: Imago, rolling upgrades and big flip. These procedures are illustrated in Figure 7.1. For the rolling upgrade and the big flip, the front-end and back-end remain shared between the old and new versions. Rolling upgrades execute for a while in a mode with mixed versions in the middle tier, while the big flip avoids this situation but uses only half of the middle-tier servers. With the former approach an upgraded node is tested online while the latter approach performs offline tests on the upgraded nodes and re-integrates them in the online system only after the flip has occurred. In contrast, Imago duplicates the entire architecture, transferring all the data objects items to $\mathbf{U_{new}}$, in order to avoid breaking dependencies during the upgrade.

Fault injection. The upgrade-centric fault model presented in Chapter 3 includes four distinct types of faults: (1) simple configuration errors (*e.g.*, typos); (2) semantic configuration errors (*e.g.*, misunderstood effects of parameters); (3) broken environmental dependencies (*e.g.*, library or port conflicts); and (4) data-access errors, which render the persistent data partially-unavailable. This fault model was created by analyzing data from three independent sources, which describes 55 individual faults (listed in Appendix B). I select 12 faults from this list (three for each fault type):

Type 1	`wrong_apache`	`config_nochange`	`config_staticpath`
Type 2	`config_samename`	`apache_satisfy`	`apache_largefile`
Type 3	`apache_lib`	`apache_port_f`	`apache_port_m`
Type 4	`wrong_privileges`	`wrong_shutdown`	`db_schema`

This selection prioritizes faults that have been confirmed independently, in different sources or in separate experiments from the same source. I inject these faults manually, during the three upgrade procedures compared in this chapter (Figure 7.1). I repeat each fault-injection procedure three times and I report the average impact, in terms of response time and yield-loss, on the system. Because this manual procedure does not accommodate injecting a large number of faults, the results must be validated using statistical-significance tests that compensate for the small sample sizes. I complement these experiments with an automated injection of randomly generated Type 1 faults.

7.1.3 Evaluation metrics

I estimate the effectiveness in performing an online upgrade, in the absence of upgrade-faults, by comparing the client-side latency of RUBiS before, and during, the upgrade for each upgrade mechanism. I assess the impact of breaking hidden dependencies by injecting upgrade faults and by measuring the effect of these faults on the system's expected availability. Specifically, I estimate the system's *yield* [Brewer, 2001], which is a fine-grained measure of availability with a consistent significance for windows of peak and off-peak load:

$$\text{Yield}(fault) = \frac{\text{Requests}_{completed}(fault)}{\text{Requests}_{issued}}$$

7.1.4 Upgrade faults and failures

An upgrade fault affects a single component and might be masked by the distributed system or by the upgrade mechanism. An upgrade failure prevents the system from correctly providing an essential functionality [Avižienis et al., 2004]. From a client's perspective, upgrade faults might cause a full outage, a partial outage (characterized by a higher response time or a reduced throughput) or a delayed outage (potentially triggered, in the future, by latent errors or security vulnerabilities). Upgrade faults might also cause the system to function incorrectly or they might have no effect at all. For example, Figure 7.2 illustrates

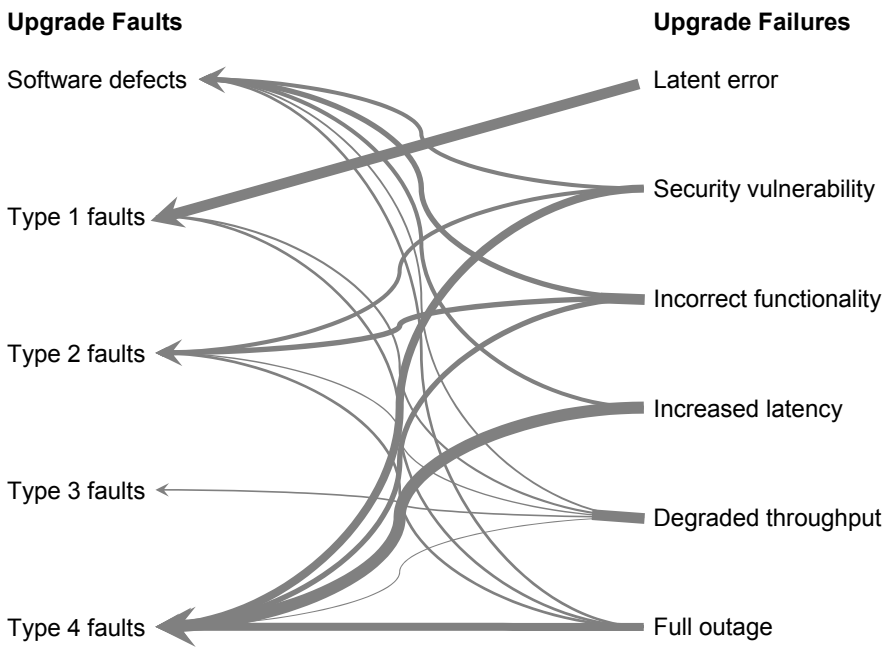


Figure 7.2. Upgrade failures and the faults that may induce them. The line thickness indicates the proportion of failures caused by a certain type of upgrade fault, according to the data sources described in Chapter 3.

the root causes for each of these failures, as reported in the three studies that provided data for the upgrade-centric fault model (see Section 3.2.3). This mapping is not universal and depends on the systems examined in the three studies. While the studies included reports of software defects, I focus on Types 1–4 of upgrade faults, which correspond to the leading cause of upgrade failures—breaking hidden dependencies.

A full outage ($Yield = 0$) is recorded when the upgrade-fault immediately causes the throughput of RUBiS to drop to zero. Latent errors remain undetected until they are eventually exposed by workload changes (*e.g.*, a peak load) or by system reconfigurations. In contrast, security vulnerabilities open up the system to external attacks but do not manifest themselves with benign system workloads. Incorrect functionality corresponds to behavior that violates the system's specification and that is not exposed by the error-reporting mechanisms of the application or of the infrastructure. Increased latency and decreased throughput are two forms of degraded functionality, which may or may not be acceptable for an upgrade mechanism. For example, during a big flip the system typically experiences a 50% drop in throughput, while one half of the system is upgraded. This outcome is expected and cannot be considered a failure.

An upgrading mechanism is able to mask a dependency-fault when the fault is detected

before reintegrating the affected node in the online system. To avoid additional approximations, I do not attempt to estimate the durations of outages caused by the broken dependencies. As the yield calculations do not include the time needed to mitigate the failures, the values reported estimate the initial impact of a fault but not the effects of extended outages.

To assess the planned-downtime requirements of each upgrade mechanism, I rely on an established benchmark for schema evolution in enterprise systems. The PRISM workbench [Curino et al., 2008a] uses the evolution of Wikipedia's database schema to evaluate automated mechanisms for performing data transformations. Additionally, Chapter 4 identifies the schema changes that commonly impose downtime for software upgrades. For the purpose of dependability benchmarking, I examine whether an upgrade mechanism supports these schema changes, in the absence of upgrade faults. I also measure the planned downtime that each mechanism imposes on its own.

7.2 Availability and overhead without faults

This section seeks to answer the question: *How much planned downtime do upgrade mechanisms impose?* A successful software upgrade may still require planned downtime (for an offline upgrade) or it may impose a high runtime overhead (for an online upgrade). As discussed in Section 4.3, complex database-schema changes, such as the ones that occurred during the upgrade history of Wikipedia, prevent rolling upgrades and impose downtime. This problem affects the big flip as well, because in this approach the database is shared between the two halves of the system. With these approaches, the system-under-upgrade is unavailable for 12 h, while the persistent data is converted to the new schema and loaded into the database of the new version.

Imago does not require planned downtime for performing the schema changes (see Section 6.1). However, Imago imposes a period of quiescence before switching to $\mathbf{U_{new}}$, by rejecting write requests at the **I** interceptors and flushing the in-progress updates to the persistent storage. Figure 7.3 shows how the duration of the flush operation varies with the system's incoming load at the time when the atomic switchover is initiated. When the middle-tier hosts are running Apache/PHP servers, the flush operation requires up to 140 s (110 s, on average, when the front-end is overloaded), including the synchronization required by the protocol from Figure 6.5. Flushing JBoss application servers requires up to

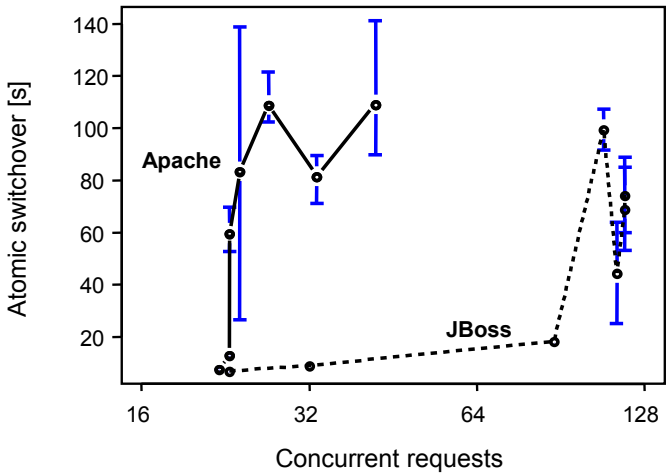


Figure 7.3. The planned downtime imposed by Imago corresponds to the time needed to complete the atomic switchover.

107 s (100 s, on average, when the front-end is overloaded). The Apache version does not accept more than 62 concurrent requests and becomes saturated faster than the JBoss version because, for Apache, the entire server is restarted in order to flush the requests to the database. For JBoss, I trigger a selective reload of the EJB application (see Section 6.2.3) involved in the upgrade. In both cases, the duration of Imago's atomic switchover does not increase indefinitely with the incoming load of client requests (note that the x-axis in Figure 7.3 is plotted on a logarithmic scale), because the front-end servers enforce admission control and do not allow a large number of in-progress requests in the system.

The switchover protocol does not cause a full outage, as the clients can invoke the read-only functionality of RUBiS (*e.g.*, searching for items on sale) while Imago is flushing the in-progress requests. Moreover, assuming that the inter-arrival times follow an exponential distribution and the workload mix includes 15% write requests [as specified by TPC-W: Menascé, 2002], I can estimate the maximum request rate that the clients may issue without being denied access. If the switchover is performed during a time window when the live request rate does not exceed 0.5 requests/min, the clients are unlikely (p=0.05) to be affected by the flush operations.

The latency of querying the content of a data item from $\mathbf{U_{old}}$ and inserting it in $\mathbf{U_{new}}$ dominates the performance of the data-transfer; less than 0.4% out of the 5 ms needed, on average, to transfer one item are spent executing Imago's code. Under a flash-crowd scenario with 1000 concurrent clients, when the site is severely overloaded, Imago must

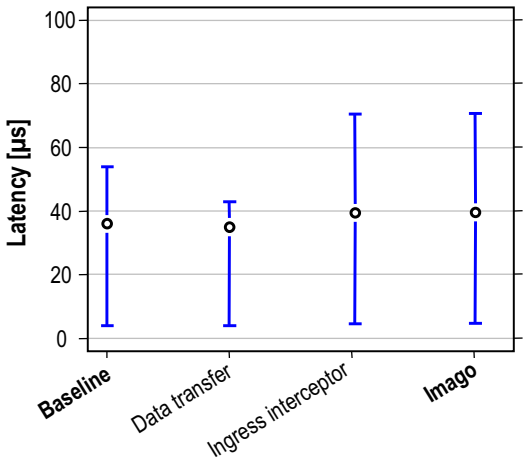


Figure 7.4. Breakdown of Imago's overhead.

make progress, opportunistically, for 2 minutes per hour in order to catch up eventually and complete the data transfer.

Figure 7.4 compares the latency overhead introduced by different Imago components (the error bars indicate the 90% confidence intervals for the RUBiS response time). The **I** interceptors impose a fixed overhead of 4 ms per request; this additional processing time does not depend on the requests received by the RUBiS front-ends.

The rolling upgrade does not impose any overhead, because sequentially rebooting all the middle-tier nodes does not affect the system's latency or throughput. The big flip imposes a similar run-time overhead as Imago because half of the system is unavailable during the upgrade (see Figure 7.5). With Imago, the upgrade completes after 13 h which is the time needed for transferring all the persistent data plus the time when access to $\mathbf{U_{old}}$ was yielded to the live workload. This duration is comparable to the time required to perform an offline upgrade: in practice, typical Oracle and SAP migrations require planned downtimes of tens of hours to several days [Downing, 2008].

7.3 Availability under upgrade-faults

This section seeks to answer the question: *How much unplanned downtime are upgrade mechanisms likely to introduce?* Table 7.1 describes the upgrade faults injected and their immediate, local manifestation. I was not able to replicate the effects of one fault (`apache_largefile`, which was reported as bugs 42751 and 43232 in the field study from Chapter 3) in the ex-

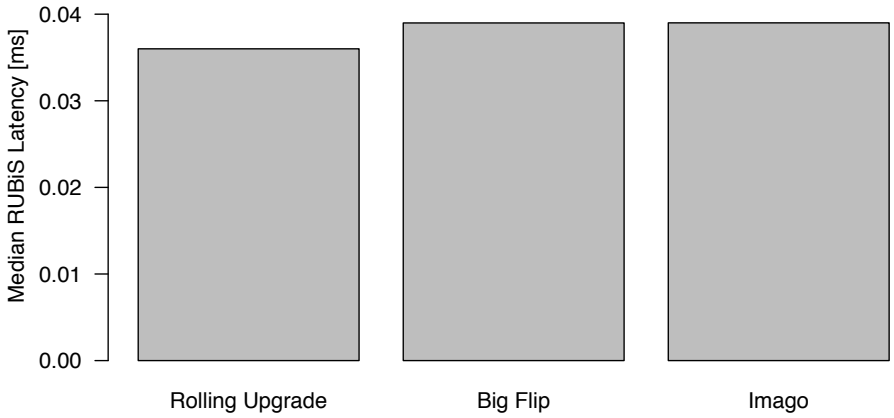


Figure 7.5. Runtime overhead imposed by online-upgrade mechanisms.

perimental test-bed. I inject the remaining 11 faults in the front-end (5 faults), middle tier (4 faults) and the back-end (3 faults) during the online upgrade of RUBiS. In a rolling upgrade, a node is reintegrated after the local upgrade, and resulting errors might be propagated to the client. The big flip can mask the upgrade faults in the offline half but not in the shared database. Imago masks all the faults that can be detected (*i.e.*, those that do not cause latent errors).

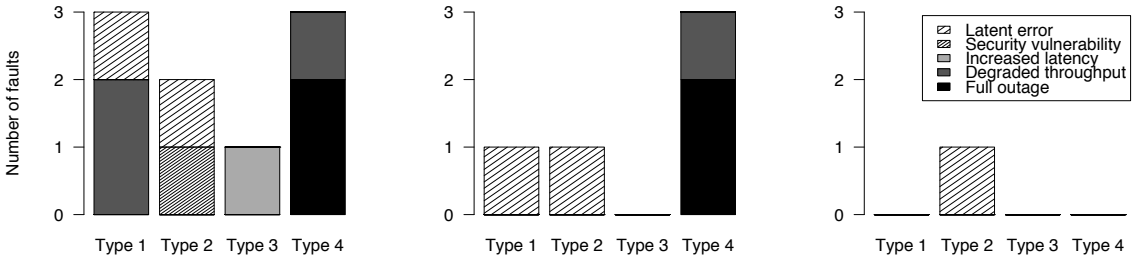
Figure 7.6 shows the impacts that Types 1–4 of upgrade faults have on the system-under-upgrade. Certain dependency-faults lead to an increase in the system's response time. For instance, the `apache_port_f` fault doubles the connection load on the remaining front-end server, which leads to an increased queuing time for the client requests and a 8.3% increase in response-time when the fault occurs. This outcome is expected during a big-flip, but not during a rolling upgrade (see Figure 7.1). This fault does not affect the system's throughput or yield because all of the requests are eventually processed and no errors are reported to the clients.

The `config_nochange` and `wrong_apache` faults prevent one front-end server from connecting to the new application servers in the middle tier. The front-end server affected continues to run and to receive half of the client requests, but it generates HTTP errors ($Yield = 0.5$). Application errors do not manifest themselves as noticeable degradations of the throughput, in terms of the rate of valid HTTP replies, measured at either the client-side or the server-side. These application errors can be detected only by examining the actual payload of the front-end's replies to the client's requests. For instance, `db_schema`

Table 7.1. Description of the upgrade faults injected.

	Name (Instances) [source]	Location	Fault-Injection Procedure	Local Manifestation
Type 1	`wrong_apache` (2) [Nagaraja et al., 2004]	Front-end	Restarted wrong version of Apache on one front-end.	Server does not forward requests to the middle tier.
	`config_nochange` (1) [Nagaraja et al., 2004]	Front-end	Did not reconfigure front-end after middle-tier upgrade.	Server does not forward requests to the middle tier.
	`config_staticpath` (2) [Nagaraja et al., 2004; Dumitraș et al., 2008]	Front-end	Misconfigured path to static web pages on one front-end.	Server does not forward requests to the middle tier.
Type 2	`config_samename` (1) [Nagaraja et al., 2004]	Front-end	Configured identical names for the application servers.	Server communicates with a single middle-tier node.
	`apache_satisfy` (1) [Dumitraș et al., 2008]	Middle tier	Used `Satisfy` directive incorrectly.	Clients gain access to restricted location.
	`apache_largefile` (2) [Dumitraș et al., 2008]	Middle tier	Used `mmap()` and `sendfile()` with network file-system.	No negative effect (could not replicate the bug).
Type 3	`apache_lib` (1) [Dumitraș et al., 2008]	Middle tier	Shared-library conflict.	Cannot start application server.
	`apache_port_f` (1) [Dumitraș et al., 2008]	Front-end	Listening port already in use by another application.	Cannot start front-end web server.
	`apache_port_m` (1) [Dumitraș et al., 2008]	Middle tier	Listening port already in use by another application.	Cannot start application sever.
Type 4	`wrong_privileges` (2) [Nagaraja et al., 2004; Oliveira et al., 2006]	Back-end	Wrong privileges for RUBiS database user.	Database inaccessible to the application servers.
	`wrong_shutdown` (2) [Nagaraja et al., 2004; Oliveira et al., 2006]	Back-end	Unnecessarily shut down the database.	Database inaccessible to the application servers.
	`db_schema` (4) [Oliveira et al., 2006]	Back-end	Changed DB schema (renamed `bids` table).	Database partially inaccessible to application servers.

causes intermittent application errors that come from all four middle-tier nodes. As this fault occurs in the back-end, both the rolling upgrade and the big flip are affected. Imago masks this fault because it does not perform any configuration actions on $\mathbf{U_{old}}$. Similarly, Imago is the only mechanism that masks the remaining Type 4, `wrong_privileges` and `wrong_shutdown`. The `apache_satisfy` fault leads to a potential security vulnerability, but does not affect the yield or the response time. This fault can be detected, by issuing requests for the restricted location, unlike the `config_staticpath` fault, which causes the front-end to serve static web pages from a location that might be removed in the future. Because this fault does not have any observable impact during the rolling upgrade or the



(a) Rolling Upgrade. (b) Big Flip. (c) Imago.

Figure 7.6. Impact of upgrade faults.

big flip, I consider that it produces a latent error. Imago masks `config_staticpath` because the obsolete location does not exist in $\mathbf{U_{new}}$, and the fault becomes detectable. The `config_samename` fault prevents one front-end server from forwarding requests to one middle-tier node, but the three application servers remaining can successfully handle the RUBiS workload, which is not computationally-intensive. This fault produces a latent error that might be exposed by future changes in the workload or the system architecture and is the only fault that Imago is not able to mask.

The rolling upgrade masks 2 faults, which occur in the middle tier and do not degrade the response time or the yield, but have a visible manifestation (the application server fails to start). The big flip masks 6 faults that are detected before the switch of the halves. Imago masks 10 out of the 11 injected faults, including the ones masked by the big flip, and excluding the latent error. A paired, one-tailed t-test[2] indicates that, under upgrade faults, Imago provides a better yield than the rolling upgrade (significant at the $p = 0.01$ level) and than the big flip (significant at the $p = 0.05$ level).

7.4 Upgrade reliability

This section seeks to answer the question: *How do upgrade faults affect the reliability of the system-under-upgrade?* Figure 7.6 suggests that broken environmental dependencies (Type 3) have little impact on enterprise-system upgrades. The manifestations of Type 3 faults (*e.g.*, a server's failure to start) are easy to detect and compensate for in any upgrading

[2]The t-test takes into account the pairwise differences between the yield of two upgrading approaches and computes the probability p that the *null hypothesis*—that Imago doesn't improve the yield—is true [Chatfield, 1983].

mechanism. Furthermore, the fault-injection results show that rolling upgrades are vulnerable to upgrade faults because the upgrade is not an atomic operation and it risks breaking hidden dependencies among the components of the distributed system. Contrary to the conventional wisdom, upgrade faults can have a global impact on the system-under-upgrade, inducing outages, throughput- or latency-degradations, security vulnerabilities or latent errors.

Compared with a big flip, Imago improves the availability because (i) it removes the *single points of failure* for upgrade faults and (ii) it performs a clean installation of the new system. For instance, the `config_staticpath` fault induces a latent error during the big flip because the upgrade overwrites an existing system. The database represents a single point of failure for the big flip, and any Type 4 fault leads to an upgrade failure for this approach. Such faults do not always cause a full outage; for instance, the `dbschema` fault introduces a throughput degradation (with application errors). However, although in this case the application error-rate is relatively low (9% of all replies), the real impact is much more severe: while clients can browse the entire site, they cannot bid on any items. In contrast, Imago eliminates the single-points-of-failure for upgrade faults by avoiding an in-place upgrade and by isolating the system version in $\mathbf{U_{old}}$ from the upgrade operations.

Imago is vulnerable to latent configuration errors such as `config_samename`, which escapes detection. This failure is not the result of breaking a shared dependency, but corresponds to an incorrect invariant of the new system, established during a fresh install. This emphasizes the fact that any upgrading approach, even Imago, will succeed only if an effective mechanism for testing the upgraded system is available.

Because this qualitative evaluation does not suggest how often the upgrade faults produce latent errors, I inject Type 1 faults automatically, using ConfErr [Keller et al., 2008]. ConfErr explores the space of likely configuration errors by injecting one-letter omissions, insertions, substitutions, case alterations and transpositions that can be created by an operator who mistakenly presses keys in close proximity to the mutated character. I randomly inject 10 typographical and structural faults into the configuration files of Apache web servers from the front-end and the middle tier, focusing on faults that are likely to occur during the upgrade (*i.e.*, faults affecting the configuration directives of `mod_proxy` and `mod_proxy_balancer` on the front-end and of `mod_php` on the middle tier). Apache's syntactic analyzer prevents the server from starting for 5 front-end and 9 middle-tier faults.

Apache starts with a corrupted address or port of the application server after 2 front-end faults and with mis-configured access privileges to the RUBiS URLs after 1 middle-tier fault. The remaining three faults, injected in the front-end, are benign because they change a parameter (the route from a `BalancerMember` directive) that must be unique but that has no constraints on other configuration settings. These faults might have introduced latent errors if the random mutation had produced identical routes for two application servers; however, the automated fault-injection did not produce any latent errors. This suggests that latent errors are uncommon and that broken dependencies, which are tolerated by Imago, represent the predominant impact of Type 1 faults.

7.5 Summary of findings

This chapter introduces a benchmark for the dependability of software upgrades, which focuses on the leading causes of planned and unplanned downtime. The impact of Type 3 faults (broken environmental dependencies) seems to be easy to detect using known techniques. Faults of Type 1, 2, and 4 frequently break hidden dependencies in the system-under-upgrade. Existing mechanisms for online upgrade are vulnerable to these faults because even localized failures might have a global impact on the system. Manual and automated fault-injection experiments suggest that Imago improves the dependability of the system-under-upgrade by eliminating the single points of failure for upgrade faults. The upgrade duration is comparable to that of an offline upgrade, and Imago can switch over to the new version without data loss and, during off-peak windows, without disallowing any client requests.

The upgrade-dependability benchmark emphasizes the availability improvements that derive from the AIR properties provided by Imago. Specifically, Isolation reduces the risk of breaking hidden dependencies, which is the leading cause of upgrade failures (see Chapter3), while Atomicity increases the upgrade reliability by avoiding system states with mixed versions. Moreover, because it does not need to maintain the synchronize the state of multiple version on-the-fly, Imago is able to perform complex schema changes that commonly require planned downtime (see Chapter 4). While not covered by this dependability benchmark, Runtime-testing allows testing states that emerge at runtime and will become important, in the future, for systems that incorporate untested components or that

Table 7.2. Trade-offs among the design choices for online upgrades in distributed systems.

	In-place	**Off-site**
Mixed versions	• Risk propagating corrupted data • Need indirection layer, with: – Potential run-time overhead – Installation downtime • Incur run-time overhead due to data conversions • Risk breaking hidden dependencies	• Risk propagating corrupted data • Need indirection layer, with: – Potential run-time overhead – Installation downtime • Incur spatial overhead
Atomic	• Incur run-time overhead due to data conversions • Risk breaking hidden dependencies • Incur spatial overhead	• Incur spatial overhead

are provisioned by third parties. Imago improves the dependability of software upgrades because it implements *dependency-agnostic upgrades*.

There are two major design choices for software-upgrade mechanisms: (i) whether the upgrade will be performed *in-place*, replacing the existing system, and (ii) whether the upgrade mechanisms will allow *mixed versions*, which interact and synchronize their states until the old version is retired. Table 7.2 compares these choices. Mixed versions save storage space because the upgrade is concerned with only the parts of the data schema that change between versions. However, mixed versions present the risk of breaking hidden dependencies; *e.g.*, if the new version includes a software defect that corrupts the persistent data, this corruption will be propagated back into the old version, replacing the master copy. Mixed, interacting versions also require an indirection layer, for dispatching requests to the appropriate version [Ajmani et al., 2006], which might introduce run-time overhead and will likely impose downtime when it is first installed. A system without mixed versions performs the upgrade in a single direction, from the old version to the new one. However, for in-place upgrades, the overhead due to data conversions can have a negative impact on the live workload. When, instead, an upgrade uses separate resources for the new version, the computationally-intensive processing can be performed downstream, on the target nodes (as in the case of Imago). Dependability benchmarking shows that in-

place upgrades introduce a high risk of breaking hidden dependencies, which degrades the expected availability.

Upgrade faults may cause several types of failures, such as full outages, throughput degradations, latency increases, *etc.* Distributed enterprise systems monitor these performance and dependability metrics using different autonomic managers, which take corrective actions when the target levels are not met. These independent actions lead to reduced ISOLATION levels, which are further explored in the next chapter.

The meadow sustains itself on a steady-state basis—unless men come along and mess it up.

Ernest Callenbach, Ecotopia, 1975

Chapter 8

Relaxing the Isolation Property: Impact Assessment for Software Upgrades

THE benchmark described in Chapter 7 suggests that the AIR properties improve the dependability of distributed systems undergoing major software upgrades. Relaxing these properties opens the door to runtime behaviors that are poorly understood and difficult to ascertain. Modern distributed systems are typically assembled from third-party components, which are optimized for the common case among many workloads, and often span multiple administrative domains. In such systems, online upgrades might disrupt the performance expectations of the critical services delivered by the infrastructure, and the interactions among multiple versions of the software expose the system to race conditions that can introduce latent errors or data corruption. In this chapter, I examine the implications of relaxed AIR properties for systems contained within a single administrative domain (*e.g.*, a private data center). Chapter 9 broadens the scope of this analysis to systems relying on components that are provisioned and administered by third parties, such as public cloud-computing infrastructures.

The Isolation property prevents a software upgrade from breaking hidden dependencies. While Imago enforces Isolation by relying on additional hardware resources (see Chapter 6), this approach might be too expensive for some systems. In these cases, where the upgrade mechanism weakens the Isolation guarantees, the impact of the upgrade on the system behavior must be assessed in advance. The scope of impact assessment extends beyond tracking functional dependencies among heterogeneous components, because the system performance and reliability also depend on the workload and vary over time. Moreover, the software upgrades might have their own deadlines for completing all the planned

changes, which must also be included in the impact assessment.

In practice, distributed enterprise systems often employ a service-oriented architecture (SOA) to isolate multiple services provided by the system. SOA aims to lower the development costs of distributed applications by allowing complex services to be composed rapidly, starting from several basic services [Papazoglou and Georgakopoulos, 2003]. Service descriptions are used to advertise the capabilities, interface, behavior, and quality of each service. An *orchestrator* implements the business logic of the distributed application, starting from a high-level specification, by coordinating the asynchronous invocation of both internal and external services [Peltz, 2003]. The expected quality of service is defined in a *service-level agreement* (SLA), which specifies several *service-level objectives* (SLOs). An SLO defines bounds and targets for a *key performance indicator* (KPI), such as response time, recovery time, availability, *etc.* An SLO also has a specific *business value* metric (*e.g.*, the penalties associated with a missed upgrade deadline or with a degraded performance) for gauging the utility of fulfilling the objective. For example, Google Inc. [2010] warrants that Gmail, and Google's other enterprise-grade services, are available at least 99.9% of the time in any calendar month. The SLA also specifies that customers will receive three days of extra service for each month when the availability drops to 99%, seven days when it drops to 95% and fifteen days when it drops below 95%.

This chapter investigates the mechanisms needed for executing dependable software upgrades in SOA. I present the design and implementation of Ecotopia, an upgrade-planning framework that takes into account the impact that such changes might have on the SOA environment. Ecotopia decouples the impact assessment (handled by multiple objective advisors, *e.g.*, performance and fault-tolerance advisors) from the change-operation scheduling (handled by an orchestrator). The orchestrator builds the upgrade schedule and estimates its impact on the business value based on the service KPIs predicted by the objective advisors.

The advisors are software components that incorporate the domain knowledge to answer "what-if" questions about service KPIs—such as performance and availability forecasts—given a description of the change operations and the timing properties associated with their execution. The orchestrator uses the advisors' predictions to compute the per-objective and aggregate business values, and converges toward an optimal upgrade schedule through an iterative refinement process. The objective advisors themselves can

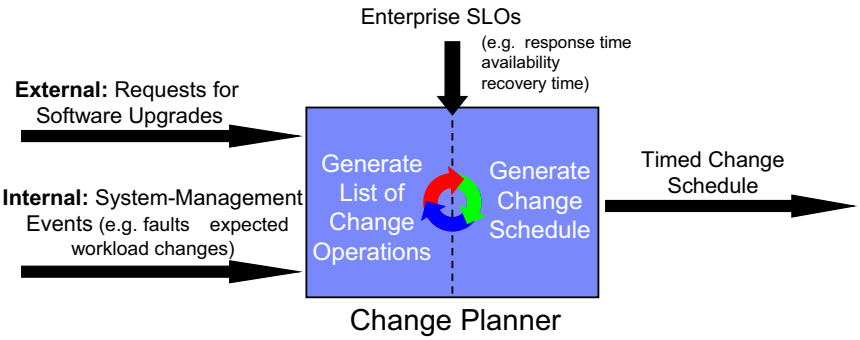


Figure 8.1. Online software upgrades, and other system changes, are likely to disrupt the critical services provided by a distributed system. Ecotopia handles changes based on both external requests (*e.g.*, software upgrades) and events detected internally by the autonomic management infrastructure (*e.g.*, faults), while taking into account their impact of the service-level objectives. The output is a timed schedule that seeks to wait for the most opportune time to apply each change operation and to maximize the enterprise business value.

be composite, third-party services.

Challenge and Contributions

While the previous chapters have shown the *effectiveness* of AIR upgrades in improving the dependability of distributed enterprise systems, I further explore the *necessity* of strong AIR guarantees. I start by focusing on the service-oriented architecture—an approach widely used for providing isolation in real-world distributed systems. The first goal of the chapter is to *understand how the guarantees provided by SOA differ from the* Isolation *property* introduced in this dissertation and the undesirable runtime behaviors that might ensue (the implications of relaxing the Atomicity property are the subject of Chapter 9).

Figure 8.1 illustrates the problem of executing software upgrades and other system changes in SOA. Some changes are planned in advance (*e.g.*, deploying new applications, upgrading obsolete software, increasing the system capacity), and are derived from an *external* request. In other cases, changes are triggered by *internal* system management events, *e.g.*, faults or load surges. Change requests include a set of partially-ordered operations and the corresponding objectives, such as the deadline for completing the change. Controlling the schedule of upgrades, to guarantee application-level correctness and to meet performance goals, remains a challenging problem [Liskov, 2001].

The scheduler must consider both the impact of the changes on all the relevant service-level objectives, as well as the objectives of each change operation. The impact assessment

should take into account the dependencies among various system components, the available prior knowledge of workload fluctuations or anticipated load surges during prime-time, as well as the degree of resource sharing across the heterogeneous, off-the-shelf components of the distributed system. This analysis should produce a timed schedule for executing the external and internal changes that are required. Therefore, the second goal of this chapter is to *identify the novel mechanisms needed for providing dependable software upgrades in SOA*, in order to illustrate the implications of relaxing the ISOLATION property.

Assumptions. I assume that KPI predictions can be derived from some knowledge of future incoming loads, either because the workloads exhibit clear trends [Arlitt and Williamson, 1996; Arlitt and Jin, 2000], or because fluctuations are preceded by recognizable patterns of warnings and notifications [Pertet and Narasimhan, 2004; Zhang et al., 2005]. Ecotopia uses the ability to predict when the system is under high and low load for optimizing across multiple service-level objectives. Furthermore, I assume that the execution times of all the change operations submitted to the planner can be estimated and that services do not have hard real-time constraints (a typical characteristic of enterprise systems).

Non-goals. This chapter does not introduce new techniques for workload prediction. Instead, I show that exploiting irregular, but predictable workloads—such as the 1998 World Cup[1] trace [Arlitt and Jin, 2000]—allows Ecotopia to improve the scheduling of change operations when pursuing multiple objectives. Moreover, Ecotopia is not designed for predicting or tolerating flash-crowd events (sudden load surges due to an unexpected increase in the site's popularity). Ecotopia's orchestrator can also cooperate with third-party advisors that answer "what-if" questions without providing workload predictions [for example: Thereska et al., 2006].

The novel characteristics of Ecotopia are:

- Rich "what-if" interaction protocol that enables the use of domain specific knowledge for an effective change scheduling decision. Unlike the previous proposals for "what-if" interactions (see Section 2.5), Ecotopia takes into account:

[1] The logs of a website dedicated to the 1998 soccer World Cup in France, containing 1.4 billion requests. The trace shows that the incoming load increases suddenly around game times, with lower peaks for the games played during weekends. This trend is typical for sites dedicated to sporting events.

  - *Timeline of prediction points*: the advisors inform the orchestrator of the expected workload changes during the scheduling timeline. The orchestrator uses these guidelines to bootstrap the scheduling algorithms.
  - *Proactive actions*: the advisors can inform the orchestrator about specific actions that may improve the impact on KPIs during related change operations. The orchestrator can include these operations in the final schedule if they result in an improved overall business-value.

- Integrated management of both *internal (*e.g., *faults, workload changes) and external (*e.g. *software upgrades) changes*. This approach is necessary because both types of changes affect a common pool of resources and services. Prior approaches [for example: Chess et al., 2005; IBM Corporation, 2007] assume different decision makers for the two types of changes.
- Optimization based on the long-term impact of change on performance and dependability objectives, accounting for both the time *during and after* execution of the change. Existing solutions consider only one of the two impact components, *e.g.*, Keller et al. [2004] consider the impact during change execution, while Anderson et al. [2002] and the Tivoli Intelligent Orchestrator [IBM Corporation, 2007] consider the impact after the change.

Section 8.1 discusses the ISOLATION level provided by SOA. Section 8.2 describes the principles of my distributed impact-assessment approach, and Section 8.3 presents the design and implementation of Ecotopia. In Section 8.4, I evaluate Ecotopia through the case study of a database upgrade in a service-oriented distributed system.

8.1 ISOLATION level provided by SOA

Structuring a distributed system as several loosely coupled services facilitates software upgrades, because most changes in the business logic translate into orchestration changes and do not require upgrading the implementation of the basic services. In these environments, the high-level service objectives are mapped into objectives for the individual system components, which are provisioned and monitored by component-specific autonomic managers [Kephart and Chess, 2003]. These managers use extensive—and sometimes proprietary—domain knowledge (*e.g.*, workload characteristics, resource utilization

models), and perform sophisticated request classification, prioritization, monitoring and request routing. As a result, *the SLOs managed by independent, third-party managers cannot always be reconciled*. For example, a workload manager prioritizes and routes the service requests by monitoring the response-time objectives, while a fault-tolerance manager primes backup nodes in anticipation of hardware faults and performs recovery by monitoring the availability objectives [Chess et al., 2005].

Moreover, *current managers tend to execute a change request as soon as possible* (*e.g.*, as soon as an upgrade is requested or a fault is detected), rather than looking for the best time to do so. For example, in a virtualized data center employing both platform management (*e.g.*, power and thermal management) and virtualization management (*e.g.*, virtual machine provisioning, SLO monitoring), the lack of coordination among managers can cause oscillatory instability between violations of the power budget and violations of the service-level objective [Kumar et al., 2009]. The platform manager reduces the CPU frequency in response to power violations, which causes SLO violations that trigger the virtualization manager to increase the CPU frequency, leading to a cycle of correlated power and SLO violations. Similarly, software upgrades performed at the wrong time can cause SLO violations because of such non-functional dependencies in distributed systems with third-party components [Dumitraș et al., 2007b; Bhattacharya and Neamtiu, 2010].

This suggests that SOA provides a relaxed version of the ISOLATION property. The complexity and the distributed nature of autonomic objective management in real-world systems makes it unfeasible for a fully centralized manager to directly assess the impact of upgrades and other change operations on each service-level objective.

Rather, *the impact on service KPIs should be estimated by the component-specific managers* that control these services. Moreover, *the managers must seek the most opportune time to execute the change operations*, based on their impact on the combined service-level objectives. Such an impact-sensitive strategy aims to respect the overall performance and dependability guarantees of the running services, yet allowing the system to incorporate upgrades and changes of various kinds.

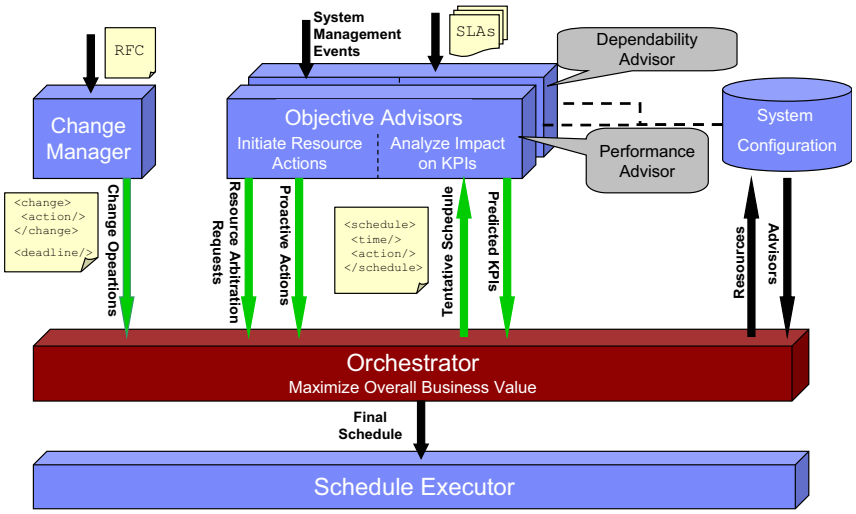


Figure 8.2. A distributed framework for impact assessment separates the tasks of impact analysis (performed by the objective advisors) and change scheduling (performed by the orchestrator). The orchestrator receives requests for change, queries the objective advisors with "what-if" questions about the tentative change schedule and uses the answers to refine the schedule with the goal of maximizing business value. The "what-if" interactions are based on an open protocol that allows the integration of third-party objective advisors.

8.2 Distributed framework for upgrade-impact assessment

The most important requirement for impact assessment in heterogeneous distributed systems, relying on third-party components, is to make minimal assumptions about the kinds of knobs that the various software components are prepared to expose to the management infrastructure. The key to achieving this goal is the separation of scheduling and impact analysis. These tasks can be performed by different components, which are provided by different vendors.

8.2.1 Framework components

Figure 8.2 illustrates the main components and interactions in our framework. The Change Manager receives high-level requests for change (RFCs), decomposes them into finer-grained change operations and related dependencies and forwards them to a centralized component, called *orchestrator*. The orchestrator receives the list of change operations and their execution constraints and generates a change schedule through an iterative process. Distributed components, called *objective advisors*, analyze the impact of change plans. The orchestrator identifies the relevant advisors by querying the System Configuration

Database. The objective advisors correspond to the autonomic component managers in the distributed system and can use domain-specific knowledge to estimate the impact of a plan on the service KPIs. The orchestrator consumes these estimations and schedules the change operations with the goal of maximizing the overall business value.

The predictions are based on detailed domain knowledge of each system component, but this knowledge is not exposed outside the objective advisors. Instead, the advisors answer simple "what-if" questions (see Section 2.5) about the impact of concrete change operations on the service KPIs, considering the workload and the tentative schedules of these operations. The orchestration is driven by the enterprise SLAs, which define methods for computing the business value [Global Grid Forum, 2004] that corresponds to the predicted KPI values. The business value reflects the utility of a given change schedule, allowing the orchestrator to compare schedules and to make impact-sensitive scheduling choices. The orchestrator sends the final schedule to the Schedule Executor, which triggers the change operations at the indicated times. The Change Manager is analogous to the Task Graph Builder from [Keller et al., 2004], and the Schedule Executor is similar to the Provisioning Manager of the Tivoli Intelligent Orchestrator [IBM Corporation, 2007]. For the rest of the chapter, I focus on the orchestrator and the objective advisors, which are novel to the distributed impact-assessment approach.

8.2.2 "What-if" interaction protocol

The interaction protocol, summarized in Table 8.1, is at the heart of this approach. As shown in Figure 8.2, a change sequence is initiated by the ChangeManager with the `InitiateChange()` function, or by an advisor with `InitiateResourceBrokering()`. The orchestrator initiates the "what-if" interaction by calling the `GetCurrentKPIs()` functions of each of the advisors to establish a baseline state for assessing the impact of the proposed schedules. Then the orchestrator creates and refines schedules through an incremental process. After arriving at a preliminary schedule of change operations, the orchestrator invokes the `GetImpactKPIs()` functions on each of the advisors to acquire the KPI predictions necessary for assessing the impact of each of the partial or complete schedules under consideration.

The advisors provide KPI estimations as time-varying functions KPI(t). I consider that a KPI value holds for a period of time, until some event causes the KPI to take another

Table 8.1. "What-if" API for distributed impact assessment.

Orchestrator	
`InitiateChange()`	Request for scheduling a group of change operations derived from a request for change.
`InitiateResourceBrokering()`	Request for reallocation of resources (*e.g.* nodes) to mitigate the impact of an event detected by the system management infrastructure (*e.g.* a hardware fault).
`ChangeSLA()`	Request for integration of SLA updates.
Objective Advisors	
`GetCurrentKPIs()`	Request for current KPI predictions for a given time interval, assuming that only infrastructure events—for instance, workload variation, node failures, but not software upgrades—will occur.
`GetImpactKPIs()`	Request for KPI predictions over a given time interval for a schedule of change operations. The reply can suggest a set of proactive actions expected to improve the KPIs in conjunction with the change operations (*e.g.* "take a database checkpoint"). Proactive actions are included in the final schedule only if they improve the overall business value.

value. This means that KPI(t) is a step function, as shown in Figure 8.3. When replying to the invocation of `GetCurrentKPIs()`, the objective advisor provides a list of pairs $< Pp_k, \text{KPI}(Pp_k) >$, indicating the times (*prediction points*) Pp_k when the KPI is expected to change and the corresponding KPI values. `GetImpactKPIs()` returns a similar list, indicating the effect of the suggested change schedule on the KPIs.

The orchestrator uses the current KPI predictions as scheduling guidelines. The initial invocation of `GetCurrentKPIs()` allows the orchestrator to learn about the prediction points, due to system-management events (*e.g.* workload surges), during the scheduling time horizon. The orchestrator then computes the business value of the current state of the system, by associating a dollar value with the various levels of service provided by the system. A service-level objective defines a target for a particular KPI. A service may have multiple SLOs (some of these objectives may track a common KPI, *e.g.* the target bounds for average latency and maximum latency), and each SLO has a business-value function.

As the KPIs can change over time, the business values are also time-variable functions.

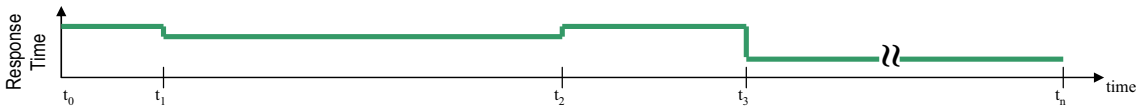


Figure 8.3. A KPI (*e.g.* average latency) varies in time, depending on the workload and the system configuration. I represent this variation by a vector of $< t, \text{KPI}(t) >$ pairs indicating the time when a KPI changes and the new value. This corresponds to a step function as shown in the figure.

If, at time t, the value of a key performance indicator is $\text{KPI}(t)$, the corresponding business value is $\text{BV}_{SLO}(\text{KPI}(t))$. For each KPI that changes at times $t_0, t_1, \ldots t_n$, the business value for the time interval $[t_0, t_n]$ is computed using a weighted average:

$$\text{BV}_{SLO}([t_0, t_n]) \quad = \quad \frac{\sum_{i=0}^{n-1} (\text{KPI}(t_i)) \cdot (t_{i+1} - t_i)}{t_n - t_0} \tag{8.1}$$

The business-value functions of different SLOs are designed to be additive. In general, the business value concept is used for reasoning about the multiple impacts of various change operations and for selecting the best trade-offs. I add the business values of all the SLOs to compute the overall business value, which reflects the utility of the proposed schedule of operations:

$$\text{BV}_{All}([t_0, t_n]) \quad = \quad \sum_{\forall SLO} \text{BV}_{SLO}([t_0, t_n]) \tag{8.2}$$

To minimize the communication costs, the orchestrator might cache the computed business value information for partial schedules. Each schedule receives a unique identifier, known to both the orchestrator and the advisors, and its related KPI predictions are saved. The orchestrator retrieves these predictions whenever it modifies the partial schedule by adding one or more change-operations, and thereby avoids repeating most of the computations. After the scheduling of a request for change is completed, the advisors add its impact on the infrastructure to the current KPI predictions.

8.3 Design and implementation of Ecotopia

Ecotopia is a concrete realization of the distributed framework for upgrade-impact assessment described in Section 8.2. Ecotopia schedules change operations with the goal of minimizing the service-delivery disruptions by accounting for their impact on the SOA environment. Ecotopia's objective advisors rely on functionality provided by component-specific

autonomic managers [Chess et al., 2005; Thereska et al., 2006; IBM Corporation, 2004, 2007; Oracle Corporation, 2005] These managers encapsulate the extensive, and sometimes proprietary, domain knowledge (*e.g.*, workload characteristics, resource-utilization models), needed for assessing the impact of change operations on the service KPIs. The implementation of Ecotopia's orchestrator is based on the Web Services standard. The orchestrator can interact with any third-party advisors that support the "what-if" interaction protocol from Table 8.1.

8.3.1 Objective-advisor implementation

The objective advisors manage separate service-level objectives (*e.g.*, performance and fault-tolerance). The advisors can be hierarchical and may span multiple administrative domains in order to manage end-to-end KPIs, in a similar manner to the resource advisor described in [Thereska et al., 2006]. The Ecotopia advisors estimate the impact of observed, predicted, or scheduled events on a few service KPIs; for instance, a performance advisor predicts violations of the response-time objectives. The predictions do not depend on the actual enterprise business-value models, which are handled by the orchestrator.

The API of the advisors contains two functions, shown in see Table 8.1. `GetCurrentKPIs()` queries the KPI predictions if changes are not applied and it is used to assess the baseline for the change impact. `GetImpactKPIs()` retrieves the KPI predictions given a tentative change-operation schedule and is used to assess the impact the change schedule. These function invocations are synchronous (*i.e.*, the orchestrator waits to receive the KPI predictions before proceeding). The reply includes the KPI predictions for the entire time horizon of the decision. This might span multiple prediction points, where the service KPIs change due to specific events such as expected workload changes or failures. The advisor reply includes one set of KPI predictions for each prediction point on the decision horizon. The replies can also suggest a set of proactive actions that are expected to improve the KPIs in conjunction with the change operations (*e.g.*, a "checkpoint database" action might reduce the expected recovery time after a fault). Proactive actions are included in the final change-operation schedule only if they improve the overall business value.

8.3.2 Orchestrator implementation

The orchestrator is a resource broker and a change–operation planner. The orchestrator starts scheduling a group of change operations in two situations (see Table 8.1): (i) `InitiateChange()` indicates that a change sequence has been initiated, following a RFC; (ii) `InitiateResourceBrokering()` indicates that a predicted or observed infrastructure event (*e.g.* a fault, a workload change) mandates a resource reassignment. All of these invocations on the orchestrator are asynchronous, *i.e.* a response containing the schedule is not provided immediately. During the scheduling process, the orchestrator communicates with the objective advisors, asking "what-if" questions in order to assess the impact of tentative change-operation schedules on the future service KPI values. The orchestrator is also invoked when an SLA has changed through `ChangeSLA()`, which indicates a modification in the overall business-value calculations. The orchestrator retrieves the new SLOs and the corresponding business-value expressions and automatically updates its scheduling engine; more comprehensive mechanisms for managing SLAs updates are described in [Roșu and Dan, 2005].

The orchestrator does not know the closed-form equation that yields the overall business value because part of this computation is performed inside the objective advisors, which act as black boxes for the orchestrator. Using the terminology of scheduling theory, the *scheduling problem has an unknown objective function* [Pinedo, 2002]. Given that the complexity of scheduling algorithms depends on their objective functions, I cannot reason about the complexity of our problem. Moreover, the closed-form expression for the business value would most likely be a non-regular objective function (a regular objective function is non-decreasing in the completion times of the change operations). There are few theoretical results for scheduling problems with non-regular objective functions. I therefore focus on approximate scheduling algorithms that make the best effort to compute a solution close to the optimal schedule.

Scheduling algorithms. The algorithms implemented in Ecotopia are based on the following pattern. Each operation e_k has a feasible scheduling interval, defined by the earliest and latest times when e_k can be scheduled, given the deadline D of the RFC and the durations d_k of the change operations:

$$\sum_{i=1}^{k-1} d_i \leq t_k \leq D - \sum_{i=k}^{n} d_i$$

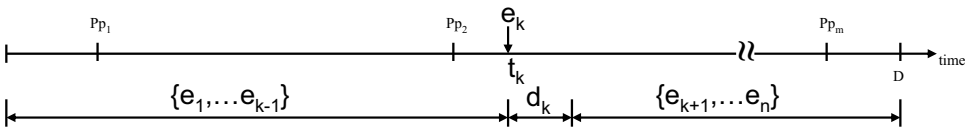


Figure 8.4. Ecotopia's greedy algorithm for scheduling change operations first chooses the change operation e_k and the time t_k that yield the best business value. This placement splits the timeline in two, and the same algorithm is applied recursively to the two halves of the problem.

Using these bounds, I try to schedule each change operation at the earliest possible time, the latest possible time and at all the m prediction points that fall within this feasible interval. The baseline scheduler uses a backtracking algorithm that generates and evaluates all of the possible placements for the change operations in an RFC. This algorithm generates the optimal schedule and has the worst-case complexity $O(m^n)$.

A more realistic scheduler uses a polynomial best-effort algorithm that is not guaranteed to provide an optimal solution. Ecotopia achieves this with a greedy algorithm: I place each operation at each possible position and we compute the resulting business value (Figure 8.4). Then, I can select either the operation and the placement that yield the best possible business value (algorithm `Greedy1`), or the operation that displays the largest overall business-value variation depending on the scheduling time, in order to avoid giving priority to the short operations that have a small negative impact (algorithm `Greedy2`). This placement splits the timeline and the change-operation group in two, and the same algorithm is applied to the two segments of the problem. These two algorithms have the complexity $O(n^2m)$ because, for scheduling each of the n operations, they evaluate nm placement options.

Schedule stability. The schedules generated by the Ecotopia remain constant in the absence of any additional change requests, SLA updates or system management events such as faults or workload changes. Figure 8.5 shows that all the changes that might affect the final schedules are always initiated outside the scheduling loop involving the orchestrator and the advisors, which ensures the stability of the schedule. The advisors generate deterministic KPI predictions for a given RFC (*i.e.*, the same tentative schedule will yield the same predictions). The predictions returned by `GetCurrentKPIs()` will be adjusted in between change groups because the effects of the change that has just been scheduled are factored into the KPI predictions; however, no such adjustment is performed inside

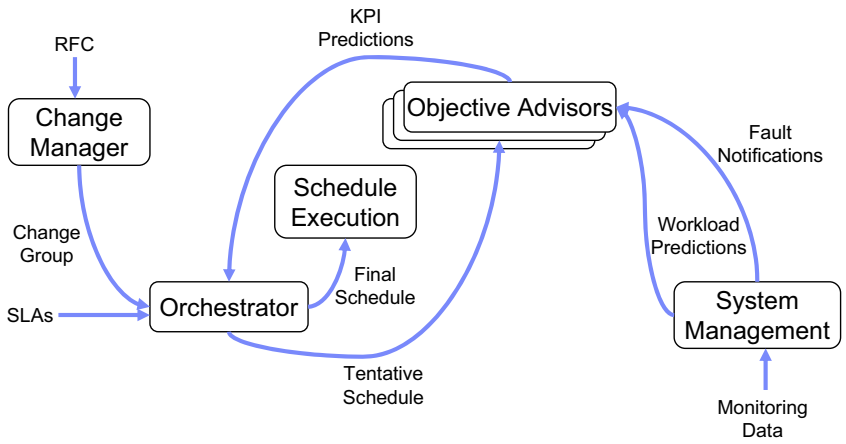


Figure 8.5. The scheduling loop of Ecotopia is designed so that all the change requests originate from outside of the iterative interaction between the orchestrator and the objective advisors. This ensures that the scheduling process does not oscillate between borderline decisions.

the scheduling loop. The algorithms presented above are guaranteed to converge if the KPI predictions are deterministic for a given change group. Other autonomic management systems based on iterative optimization loops [Anderson et al., 2002; Golding and Wong, 2006] may oscillate between borderline decisions because a resource reconfiguration will affect the performance metrics which may subsequently trigger another reconfiguration. Ecotopia, where all of the changes are initiated outside the scheduling loop and the "what-if" analysis considers a long time-horizon, guarantees that such infinite cyclic dependencies are broken and that oscillations cannot occur.

8.4 Case study: Software upgrades a service-oriented enterprise system

I consider a two-tiered system, where the physical hosts are organized in independently managed node-groups. The first tier is a node group of application servers managed by application server middleware [for example: IBM Corporation, 2004] and the second tier is a node group of database servers, managed by a clustering infrastructure [for example: Oracle Corporation, 2005]. The two node-group managers perform various middleware-specific management tasks (*e.g.*, load balancing, request routing, fault recovery). This infrastructure provides two services, each mapped onto corresponding application-server and database services. The two services processing Web transactions are load-balanced

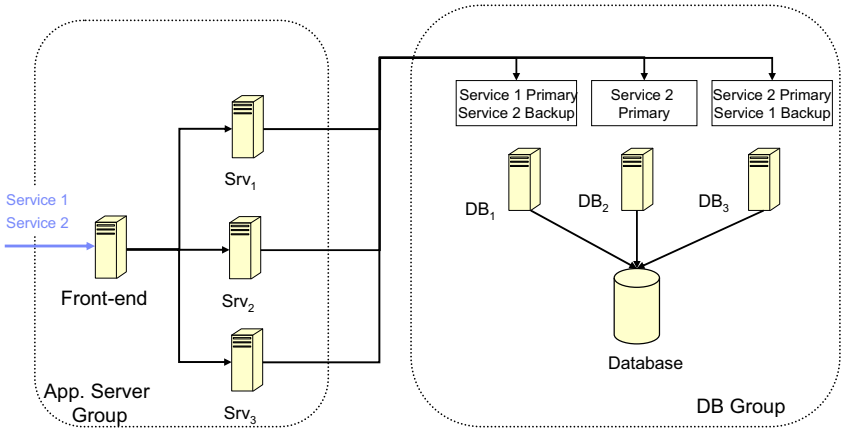


Figure 8.6. Sample two-tier system managed by Ecotopia.

across three application servers, W_1 to W_3. These front-end services query two database services that connect to separate database partitions. The database group comprises three nodes:

- DB_1 acts as primary server for $Service_1$ and as backup for $Service_2$;
- DB_2 is part of the logical primary server for $Service_2$, which is distributed on two database nodes;
- DB_3 is also part of the logical primary for $Service_2$ and it is a backup for $Service_1$ as well.

Each of the two enterprise services has response time, recovery time and availability objectives. The business value associated with these SLOs depends on the related KPIs, such as the total number of transactions or the number of transactions with response time below target.

A performance advisor evaluates the impact of change operations on the end-to-end response time for each service by exploiting the knowledge provided by the node-group managers (*e.g.*, expected workload variations, service overheads). Similarly, a fault-tolerance advisor evaluates the impact on the recovery time and the availability SLOs. I discuss a realistic scenario of upgrading the database software (Section 8.4.1), and I complement this analysis with measurements illustrating the trade-off between the cost and the loss of optimality of different scheduling algorithms (Section 8.4.2).

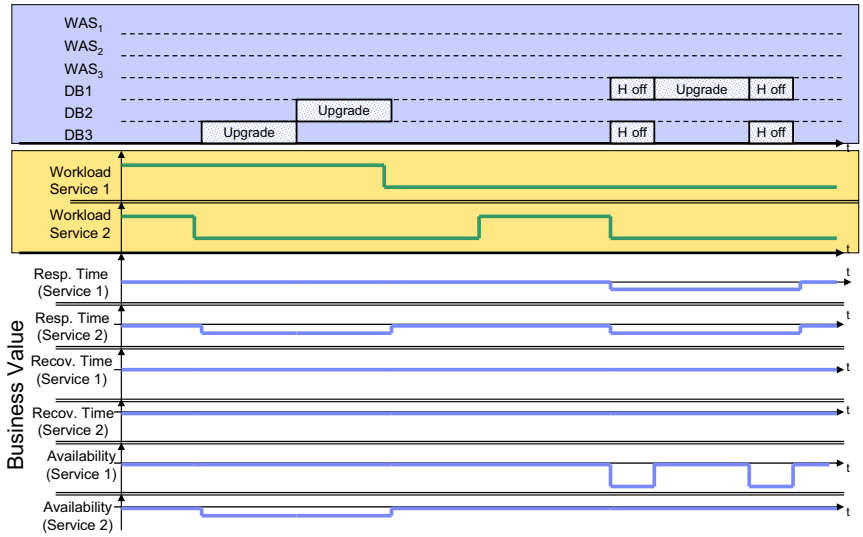
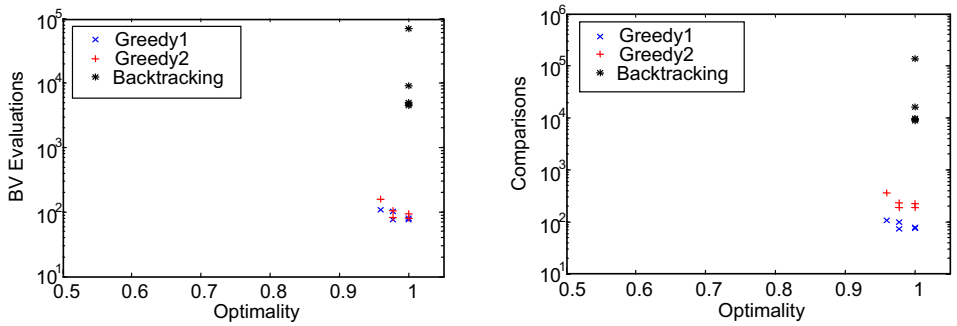


Figure 8.7. Database upgrade scenario.

8.4.1 Qualitative evaluation

Ecotopia decomposes the RFC requesting a database upgrade into finer-grained change operations: each database node is upgraded separately and, for upgrading DB_1, $Service_1$ is handed off to DB_3 (its backup) before the upgrade and restored at the end. The analysis must consider the impact of these operations on service objectives and their corresponding business values. For instance, if the load on $Service_1$ is high (see Figure 8.7), Ecotopia can reorder the change operations to perform the upgrades on nodes DB_2 and DB_3, which are used by $Service_2$. In fact, the upgrade of DB_1 must be delayed until both services register low incoming request rates because a high request rate during the upgrade may overload DB_3, which also handles both $Service_1$ and $Service_2$. By delaying the upgrade, the penalties incurred for violating the response time objectives are minimal, thus maximizing the aggregate business value for the duration of the upgrade. The reordering must take into account the functional dependencies between change operations; thus, the hand-offs of $Service_1$ should precede and follow the upgrade of DB_1.

Scenarios such as this one are typical of change management in distributed enterprise systems. Similar operations occur at a much larger scale in many real-world deployments. This scenario shows that delaying the change operations may sometimes improve the overall business value. This illustrates the complexity of predicting the impact of change due to the strong dependencies on the actual implementations of objective managers. Ecotopia



(a) Number of business-value evaluations. (b) Number of business-value comparisons.

Figure 8.8. The scheduling algorithms implemented in Ecotopia choose different trade-offs between the cost of scheduling and the loss of optimality. The greedy algorithms are polynomial and yield schedules with a business value within 95% of the optimal achievable business value, which is computed using the exponential backtracking algorithm.

addresses these issues by delegating the impact assessment to component-specific advisors that encapsulate all the relevant domain knowledge.

8.4.2 Quantitative evaluation

Using a traditional scheduler, which does not optimize for long-term impact [for example: IBM Corporation, 2007; Keller et al., 2004; Anderson et al., 2002; Golding and Wong, 2006] would result in executing all of the change operations as soon as possible, instead of waiting for the most opportune time when the incoming load is low. The outcome of such impact-insensitive scheduling is a missed opportunity for optimizing the overall business value. Instead, the scheduling algorithms presented in Section 8.3.2 find the optimal schedule for these two scenarios, and the run-times of all the algorithms—including the exponential backtracking scheduler—are comparable (less than 1 s). I also test the scheduler using several randomly-generated input sets, and I explore the trade-off between complexity and the loss of optimality. The most appropriate complexity measure is the number of times the business value needs to be evaluated, since these evaluations require communication between the orchestrator and the advisors. The loss of optimality shows how close the business value of the resulting schedule was to the BV of the optimal schedule, as generated by the backtracking algorithm. Figure 8.8 shows that, for small problems (*e.g.*, 5 change operations and 10 KPI prediction points), the two (polynomial) greedy algorithms obtain

near-optimal results and they need one or two orders of magnitude fewer BV evaluations than the exponential, optimal backtracking algorithm.

For larger problems, we cannot use the backtracking algorithm and, therefore, we cannot measure the loss of optimality of the greedy schedulers. For 100 change events and 100 prediction points, the greedy algorithms requires up to 36673 business-value evaluations and 67342 comparisons, sometimes with significant differences between the two algorithms (between 3% and 68%). `Greedy1` also exhibits a higher variance of the number of business-value evaluations than `Greedy2`. While a scenario where `Greedy2` performs better than `Greedy1` could easily be constructed, the two algorithms produced identical schedules for all but one of the randomly-generated input traces.

8.5 Summary of findings

This chapter shows that the service-oriented architecture provides a relaxed version of the ISOLATION property. While SOA is widely used for isolating different services in real-world distributed systems, it is unable to guarantee some non-functional dependencies, such as performance levels that can be affected by several autonomic managers. The interplay of change management (*e.g.* software upgrades) and workload variability requires additional mechanisms for ensuring the dependability of distributed systems using SOA. Ecotopia tackles the complexity and the distributed nature of SLO management in real-world systems by separating the impact assessment (performed by the objective advisors) from the scheduling and business-value computation and aggregation (performed by the orchestrator).

By focusing on the "what-if" interaction protocol rather than on building a monolithic change-management system, Ecotopia facilitates changes that might span target heterogeneous software infrastructures. The orchestrator can communicate with third-party advisors, which are built with specific, proprietary domain knowledge about a service/system/vendor, and construct schedules using only the information available from such advisors. This approach mirrors the philosophy of SOA, which is to focus on interaction protocols rather than on implementation bindings.

The separation between scheduling and impact assessment may limit Ecotopia's optimization capabilities when the advisors cannot provide a comprehensive impact analysis

(*e.g.*, some third-party managers do not provide latency estimations, which are required for end-to-end response-time management). Moreover, KPI predictions typically have a degree of inaccuracy, especially when the time frame of the predictions is far ahead in the future. If the advisors provide incorrect information, the orchestrator might take the system to a state with unacceptable service levels; in this case, a downgrade or the rollback of the changes can be scheduled using the same process described above.

Cine fuge după doi iepuri, nu prinde niciunul.
A man who chases two rabbits catches none.

Romanian Proverb

Chapter 9

Relaxing the ATOMICITY Property: Mixed-Version Race Conditions

CHAPTER 8 has shown that the service-oriented architecture provides a relaxed version of the ISOLATION property and requires additional mechanisms for ensuring the dependability of software upgrades. Additionally, in systems spanning *multiple administrative domains—e.g.*, Web applications that rely on client-side code or enterprise systems that lease cloud-computing resources—the ATOMICITY property cannot always be enforced. When the enterprise does not control all the tiers of the system and cannot plan and coordinate the upgrade process, the system is exposed to race conditions that can introduce latent errors or data corruption.

Online upgrades that do not enforce ATOMICITY 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. Supporting multiple versions that operate concurrently and that keep their states synchronized requires a careful coordination of the upgrade process, as illustrated in the industrial best-practice recommendations for performing rolling upgrades [Microsoft Corporation, 2005; Oracle Corporation, 2008; see also Section 2.3.4] . In general, however, 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 [Segal, 2002].

This chapter introduces a new kind of race condition, involving multiple versions of the software, that has not been described before in the research literature. Such *mixed-version races*[1] can occur, during rolling upgrades, in systems that communicate across administra-

[1] Mixed-version races were first reported in [Dumitraș et al., 2010], which includes a preliminary discussion of the results from this chapter.

tive domains using asynchronous messaging. Mixed-version races might be benign (they occur frequently during Facebook upgrades [Reiss, 2009]), 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 [Hansell, 1994]).

This prior anecdotal evidence suggests that some real-world upgrade failures can be traced back to mixed-version races. Conversely, delaying the upgrade of a system with known software defects (*e.g.*, bugs or missing features) might also have a negative impact. The trade-offs between upgrading and not upgrading are not easy to ascertain.

Software upgrades that provide the ATOMICITY property or that are performed offline are not affected by mixed-version races. When these, or other techniques for preventing mixed-version races, are too expensive or infeasible, the system administrators must take a different approach to prevent system failures. This chapter describes a comprehensive model that takes 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. By assessing the impact of bugs and mixed-version races on the system, this analytical model allows system administrators to quantify and compare the risk of following an online-upgrade plan with the risk of delaying or canceling the upgrade.

I 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, I assess their impact in a limited number of system configurations and I derive the overall risk of upgrading analytically. While it is challenging to determine what humans would find useful for carrying out system administration tasks, I point out that the risk model commonly recommends counter-intuitive, but correct, decisions. Risk assessment represents a method of last resort, for the situations where mixed-version races cannot be avoided through other technical means.

Challenge and Contributions

While Chapter 8 has discussed the implications of relaxing the ISOLATION property during an online upgrade, this chapter further explores the necessity of strong AIR properties by focusing on distributed systems that span multiple administrative domains, where upgrade

ATOMICITY is difficult to enforce. Specifically, the goal of this chapter is to *assess the worst-case impact of performing software upgrades with relaxed levels of* ATOMICITY.

The ATOMICITY property ensures that clients can access only one version—either the new one or the old one—during the software upgrade. However, large Internet systems, such as Google, Facebook or Wikipedia, often employ rolling upgrades, which do not guarantee ATOMICITY and create system states with mixed versions (see Section 2.3.4). While the new version can be backward-compatible, the old version can not handle invocations that require the new version's semantics. Prior approaches for upgrading in the presence of mixed versions advocate upgrading the servers before their clients [Kramer and Magee, 1985; Segal and Frieder, 1989b; Tewksbury et al., 2001], to prevent the new version from calling into the old version, or simulating the interfaces of past and future versions during the upgrade [Ajmani et al., 2006].

These approaches are infeasible in distributed systems that communicate across multiple administrative domains, where an online upgrade's administrator does not control all the tiers and cannot coordinate their upgrades. For example, Tewksbury et al. [2001] observe that certain communication patterns used in practice—such as one-way or asynchronous messages—prevent enforcing the quiescence needed for upgrading the components that receive these messages.

Instead of preventing race conditions, or other undesirable behaviors that can result from an online upgrade lacking ATOMICITY, I propose assessing the risk they pose to the system. This means asking 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 [Swanson, 1976], *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.

Assumptions. The risk assessment approach described in this chapter assumes that the software developers and system administrators use a uniform labeling system, which covers the severity of known defects, the criticality of feature addition/change/removal requests, as well as the severity of the inconsistencies that might result from mixed-version races. Secondly, I assume that a thorough integration-testing procedure is in place, and that it can be extended to the system states with mixed versions. Thirdly, I 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. Finally, I assume that the processing time of a user request is negligible compared to the time needed to upgrade a host within the distributed system. These assumptions guide the derivation of the analytical risk model, described in Section 9.2.

Non-goals. This chapter does not propose novel techniques for preventing or masking the effects of mixed-version races. Imago, for example, enforces upgrade Atomicity (see Chapter 6), which prevents mixed-version races. However, because similar techniques might be infeasible under the realistic assumptions of the systems targeted in this chapter, I present the best possible alternative: risk assessment. The risk assessment focuses on the impact of mixed-version races and does not consider the possibility that the new version might also include known software defects. 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 chapter. Moreover, this chapter does not seek to assess the accuracy of the information produced by the risk assessment. While the upgrade-centric fault model from Chapter 3 provides a window into the reasons why software upgrades fail, the lack of data on upgrades across multiple administrative domains currently prevents a quantitative validation of the upgrade-risk model.

This chapter makes two original contributions:

- I identify *mixed-version races*, which can occur during rolling upgrades in distributed systems spanning multiple administrative domains, and I describe the system interactions that lead to such race conditions (Section 9.1). While mixed-version races have not been characterized before, two real-world examples of upgrade failure [Hansell, 1994; Reiss, 2009] can be traced back to this type of race condition. Mixed-version races are enabled by the absence of upgrade Atomicity.
- I develop an *analytical model* for reasoning about the trade-off between upgrading in the presence of mixed-version races and delaying an upgrade that corrects known software defects (Section 9.2). Unlike the previous approaches for evaluating the dependability of online upgrades [for example: Oppenheimer et al., 2003; Oliveira et al., 2006; Crameri et al., 2007; Zheng et al., 2009; see also Chapter 7], this analytical model does not rely on field or experimental data. Instead, the risks of software

upgrades are estimated from system parameters and testing results that are readily available to the developers and administrators. I also demonstrate, on three case studies, how this model can be used to make informed decisions regarding whether *to upgrade or not to upgrade* (Section 9.3). The goal of this qualitative evaluation is to determine whether the risk model provides the upgrade administrators with useful information, which cannot be obtained through other means.

9.1 Mixed-version races

In practice, rolling upgrades are widely believed to reduce the risks of upgrading because failures are localized and might not affect the entire distributed system [Oracle Corporation, 2008; Downing, 2008]. However, rolling upgrades also introduce the risk of race conditions between the old and the new versions of the software. 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 an invocation from the new version is processed by the old version on a different tier of the application.

I illustrate mixed-version races with an online banking example. Banks are starting to employ online upgrades [Choi, 2009], in spite of the inherent risks of data inconsistency associated with current upgrading approaches. I 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 9.1):

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.

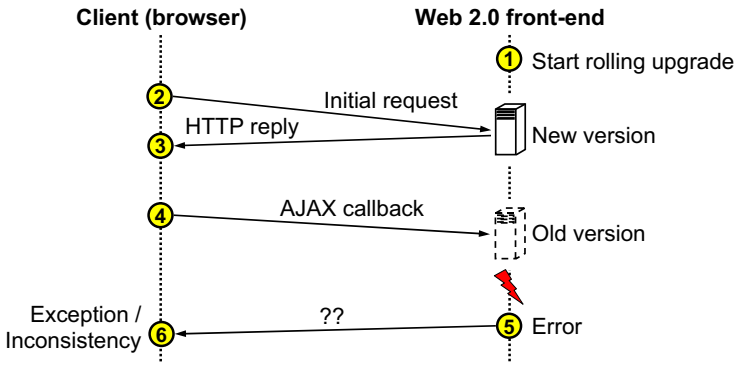


Figure 9.1. Anatomy of a mixed-version race.

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.

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 [Reiss, 2009].

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 $1 from a bank account would subtract $2 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 [Hansell, 1994].

9.1.1 Key technical challenges

The mixed-version race described above could have been prevented 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. However, in a large enterprise infrastructure some servers are likely to become unresponsive during the upgrade—either because they have failed or because they are slow to upgrade—which makes it difficult to determine reliably when the rolling upgrade has completed [Hume, 2010]. Prior anecdotal evidence, from the recorded occurrences of mixed-version races [Hansell, 1994; Reiss, 2009], confirms that these race conditions cannot be avoided easily.

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

- **Upgrades lacking** Atomicity**.** 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 asyn-

chronous I/O. Asynchronous communication is considered by some experts a better paradigm for building distributed systems than synchronous RPC [Vinoski, 2008].

- **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 distributed enterprise systems, similar problems will become widespread. Distributed systems have been using heterogeneous, off-the-shelf components for a long time (see Chapter 8). 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.

9.2 Upgrade risk model

The risk 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—I estimate the *risk of upgrading*. I 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, I estimate the expected impacts of the two alternative decisions—to upgrade or not to upgrade—over the typical time frame of a rolling upgrade. The key idea is to treat mixed-version races as software defects, by incorporating them in the regular software testing activities.

9.2.1 Integration in the software-development life cycle

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. Software engineering textbooks [for example: Sommerville, 2007] recommend thoroughly planning the changes offline, taking into account the characteristics of the whole system. After the changes have been implemented, they are typically integrated in a copy of the running system and tested, *e.g.* by running a regression test suite, before deploying them in the production system.

Similarly, mixed-version testing can be done using only two hosts, one running the new version and the other running the old version, by triggering the worst-case scenario leading to a mixed-version race: a callback from the new version that arrives at the old version, as described in Section 9.1. 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 (see also Section 4.1). This could lead to $2^{120} \approx$ 1E36 (one undecillion) possible version combinations during a rolling upgrade similar to the one described in Section 9.1. Instead, I test only one combination.

This testing approach can be extended 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, all the potential interactions where a version invokes an older version must be considered, *i.e.* $\binom{n}{2} = \frac{1}{2}n(n-1)$ mixed-version combinations. In the second case, all the interactions where the new version invokes the old version in the next tier must be considered, *i.e.* $(m-1)2^{m-2}$ 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 up-

grade 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 chapter, I 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.

In most cases, developers and administrators cannot estimate accurately the likelihood of exposing known software defects or the variability of upgrade durations for each host. To enhance the usability of the analytical risk model, I use a discrete probability measure, with three possible values: low, medium, and high. Similarly, the risk model requires 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.

9.2.2 Analytical risk model

Table 9.1 describes the input and output parameters of the 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. I assess:

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

which combine the likelihoods of inconsistencies and bug manifestations with the corresponding severity levels. I 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

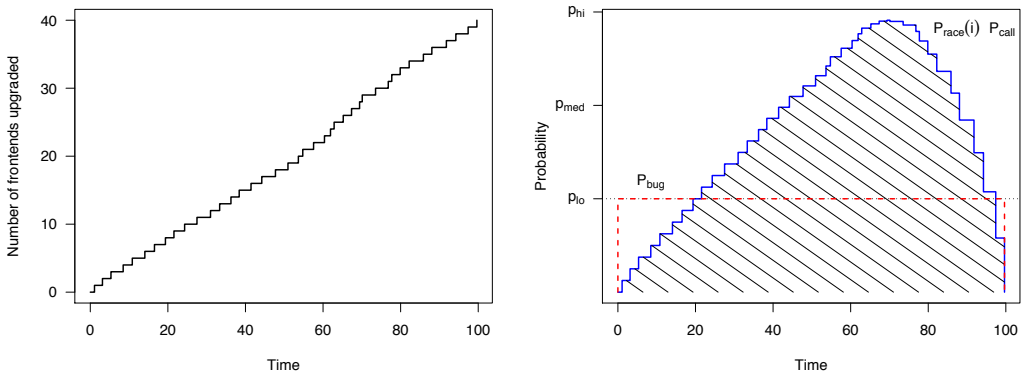
Table 9.1. Summary of notations from the upgrade risk model.

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, I 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.

than precise probability values. In this analysis, I do not attempt to assign placeholder values to these probability levels, and instead I derive the risk symbolically. To avoid counter-intuitive artifacts in the computation, I consider that p_{lo}, p_{med} or p_{hi} correspond to a linear scale, *i.e.* $p_{med} = 2p_{lo}$ and $p_{hi} = 3p_{lo}$.

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



(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 9.2. Analytical risk model, comparing the expected impacts of upgrading and of not upgrading.

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:

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

The probability of mixed-version races P_{race} varies during the upgrade. I 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 9.2a. I do not assume that durations τ_i are known precisely when planning the upgrade. However, I 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}). I 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 9.1), and sending any of the subsequent callbacks to the old version (event $\mathbb{E}_{2.2}$, analogous to step 4 in Figure 9.1):

$$P_{race}(i) = \Pr[\mathbb{E}_{2.1} \text{ at } t_i] \cdot \Pr[\mathbb{E}_{2.2} \text{ at } t_i]$$

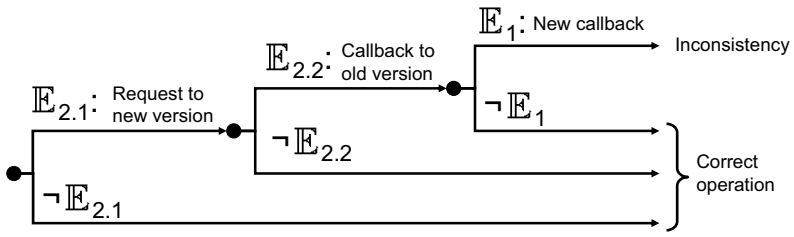


Figure 9.3. Events leading to a mixed-version inconsistency.

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

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

$$\begin{aligned}\Pr[\mathbb{E}_{2.2} \text{ at } t_i] &= 1 - \left(\frac{i}{U}\right)^c \\ P_{race}(i) &= \frac{i}{U} \cdot \left(1 - \left(\frac{i}{U}\right)^c\right) \qquad (9.1)\end{aligned}$$

$P_{race} = 0$ 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 9.2b illustrates the evolution of P_{race} and P_{bug} during the rolling upgrade. The shaded area under the bell curve corresponds to the likelihood of mixed-version races during the upgrade, and the dashed rectangle corresponds to the expected occurrence of bugs during normal operation.

I 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 9.3. After the upgrade of the i^{th} server, the risks of upgrading and of not upgrading are:

$$\mathbf{Risk}_{\text{no upgrade}} = \frac{\sum_{k=1}^{N_{bug}} P_{bug}(k) \cdot \mathbb{S}(\mathbb{B}_k)}{N_{bug} \cdot \max \mathbb{S}} \qquad (9.2)$$

$$\mathbf{Risk}_{\text{upgrade}}(i) = \frac{i}{U} \cdot \left(1 - \left(\frac{i}{U}\right)^c\right) \cdot \frac{\sum_{k=1}^{N_{call}} P_{call}(k) \cdot \mathbb{S}(\mathbb{I}_k)}{N_{call} \cdot \max \mathbb{S}} \qquad (9.3)$$

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 $\mathbf{Risk}_{\mathbb{D}} \in [0, 3p_{lo}]$. I consider

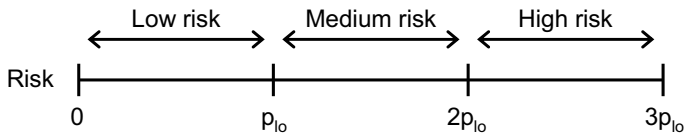


Figure 9.4. Discrete risk values.

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 9.4).

The average risk of upgrading is:

$$\overline{\mathbf{Risk}_{\text{upgrade}}} = \frac{\sum_{i=1}^{U} \tau_i \cdot \mathbf{Risk}_{\text{upgrade}}(i)}{t_U} \tag{9.4}$$

This formula does not have a closed-form expression in terms of $\overline{\tau}$, τ_{lo} and τ_{hi}. Instead, this risk can be estimated 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, I also compute the 95% confidence interval for the average risk of upgrading, which indicates the precision of the estimation.

The maximum risk of upgrading, however, can be computed using a simple, closed-form expression. I compute this maximum by approximating the probability of sending a new callback to the old version, from Equation 9.1, 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]

$$\max(P_{race}) = \sqrt[c]{\frac{1}{c+1}} \cdot \left(1 - \frac{1}{c+1}\right) \tag{9.5}$$

[2]This formula computes an upper bound, because the $P_{race}(x)$ is a stair function and $P_{race}(x) \leq \tilde{P}_{race}(x)$. 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 9.1.

The maximum value $\max(P_{race})$ depends only on c, and its asymptotic bound is 1. However, for typical values of c, this 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	7	8	9	10
$\max(P_{race})$	0.25	0.38	0.47	0.53	0.58	0.62	0.65	0.68	0.70	0.72

The maximum risk of upgrading is:

$$\max(\mathbf{Risk}_{\text{upgrade}}) \quad = \quad \max(P_{race}) \cdot \frac{\sum_{k=1}^{N_{call}} P_{call}(k) \cdot \mathbb{S}(\mathbb{I}_k)}{N_{call} \cdot \max \mathbb{S}} \tag{9.6}$$

A computer program that automates these calculations is included in Appendix C.

9.2.3 Interpretation

The 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. I 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 administrators 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 I 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 services 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.

9.3 Qualitative validation of the analytical risk model

Complete data on real-world upgrade failures is scarce and hard to obtain, due to the sensitivity of this subject. Two real-world examples of upgrade failures can be traced to mixed-version races [Hansell, 1994; Reiss, 2009]. Because, to the best of my knowledge, this race condition has not been characterized before, the anecdotal information available does not provide sufficient data to design statistically significant experiments for evaluating the risk of upgrading in the presence of mixed-version races. Moreover, the 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.

I conduct a qualitative evaluation of our risk model, seeking to answer the question: *Is this risk model useful?* By walking through three hypothetical—but realistic—scenarios of online upgrades, I 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 9.3.1, and foreign exchange, in Section 9.3.3) and one focuses on a large-scale system that is not mission critical (a social networking site, in Section 9.3.2). I show that using the 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. The 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 chapter, I do not ask the question: *How accurate is the additional information provided by the risk model?* This question could be answered by using the model in a production

system, for an extended period of time, and by reporting on this experience after observing real upgrade failures. I believe that such practical experience is essential for providing a complete validation of the risk-assessment approach.

9.3.1 Upgrade #1: Online banking

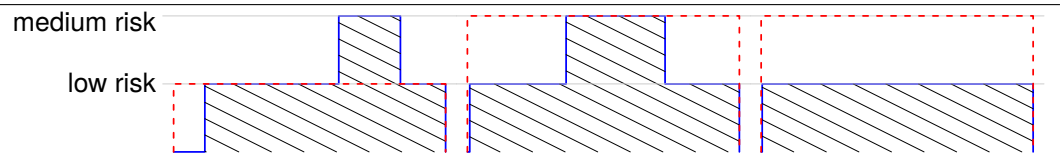
Imagine that a bug in the Web interface of an online banking application (such as the one described in Section 9.1) was reported and corrected. Specifically, in the old version, an edit box for entering fund transfer information accepts all alpha-numeric characters rather than restricting user input 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 [CWE/SANS, 2010]. 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 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 9.2).

The analytical model allows me to 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 intuition, the analytical model predicts that it is better to upgrade during a planned main-

Table 9.2. To upgrade or not to upgrade? Comparison of risk predictions in three realistic scenarios of online upgrades. The sparklines in the bottom row illustrate the time-variable risk of upgrading and the constant risk of not upgrading.

	Online banking §9.3.1	Social networking §9.3.2	Foreign exchange §9.3.3
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
c	6	2	1
$\max \mathbb{S}$	5	5	5
N_{call}	1	1	1
$P_{call}(1)$	p_{hi}	p_{hi}	p_{med}
$\mathbb{S}(\mathbb{I}_1)$	3	5	3
N_{bug}	1	1	1
$P_{bug}(1)$	p_{lo}	p_{med}	p_{hi}
$\mathbb{S}(\mathbb{B}_1)$	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
			

tenance 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.

9.3.2 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.

9.3.3 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. The analytical model shows that the risk of upgrading is always lower than the risk of not upgrading, and recommends an online upgrade. The analytical model provides a systematic approach for deciding whether to upgrade or not to upgrade.

9.4 Summary of findings

This chapter describes 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. I also introduce an analytical model 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.

The 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, Sliwerski et al. [2005] 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 [Microsoft Developer Network, 2001] and Facebook [Reiss, 2009] also deploy their upgrades on Tuesdays.

Recent advances in low-overhead dynamic analysis [for example: Liblit et al., 2003; Bond et al., 2010] 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 distributed-system upgrades are errors in the upgrade procedure, rather than software defects (see Chapter 3). Moreover, these failures are often hard to replicate outside of the deployment environment, because they correspond to broken dependencies, they affect systems spanning multiple administrative domains and their manifestations are workload-dependent. The current lack of standard benchmarks for dependable software upgrades makes it challenging to assess the terms of Equation 9.3, such as the severity of mixed-version inconsistencies or the number of hosts involved in the upgrade in order to compute the risk of upgrading ($\mathbf{Risk}_{\text{upgrade}}$). Chapter 7 discusses the first results in dependability benchmarking for software upgrades, but further investigation is needed, in particular regarding upgrades that span multiple administrative domains.

Mixed-version races are an example of behavior that emerges at runtime and that cannot be tested, exhaustively, before the system is deployed. Online software upgrades interact with the workload in ways that may be unpredictable at design-time. Reasoning about such emerging behavior is difficult, and this chapter represents a first step toward a systematic approach for validating such runtime-emerging behaviors.

Chapter 10

Conclusion

THIS dissertation identifies and addresses the leading causes of both unplanned failures and planned downtime resulting from software upgrades in distributed systems. This dissertation defines three abstract properties required for improving the dependability of software upgrades. Building on empirically derived insights on current upgrade practices and problems, this approach harnesses the opportunities provided by emerging technologies such as cloud computing to simplify large-scale upgrades, to allow upgrades to be executed efficiently online, and to improve their dependability.

10.1 Summary

Traditional fault-tolerance approaches concentrate almost entirely on responding to, avoiding, or tolerating unexpected faults or security violations. However, scheduled events, such as software upgrades, account for most of the system unavailability and often introduce data loss, latent errors or race conditions. I take a holistic approach and focus on upgrading distributed systems end-to-end.

Industry trends suggest that online upgrades are currently needed in large-scale systems, such as electrical utilities, assembly-line manufacturing, customer support, e-commerce, banking, *etc.* However, previous research has focused on upgrading individual system components of distributed systems (*e.g.*, the application code, the middleware framework or the database schema). Similarly, industrial best practices recommend a gradual deployment of upgrades, which place the system in a state with mixed, interacting versions.

Through two empirical studies, I identify the leading causes of upgrade failure—breaking hidden dependencies—and of planned downtime—changing the format of persistent data. I establish an upgrade-centric fault model, by analyzing multiple sources of fault data through statistical clustering techniques that are widely used in the natural sciences for creating taxonomies of living organisms. Most upgrades fail because of unavoidable human errors in the upgrade procedure, which break dependencies in the system under upgrade (*e.g.*, by specifying wrong service locations, creating database-schema mismatches, introducing shared-library conflicts). I also identify incompatible schema changes and computationally intensive data conversions as the leading causes of planned downtime for Wikipedia, a popular Internet system.

This dissertation introduces the AIR properties—Atomicity, Isolation and Runtime-testing—which improve the dependability of software upgrades by removing the leading causes of planned and unplanned downtime. The Isolation property provides an alternative to tracking dependencies. By accessing the old version in a non-intrusive, read-only manner, I avoid breaking hidden dependencies during the upgrade. The Atomicity property implies that the system must not include mixed versions, and the Runtime-testing property ensures that the upgrade does not fail because of differences between the testing and deployment environments. These properties enable long-running data conversions in the background, during an online upgrade, as the new version is inactive, and does not need to be in a consistent state, until the atomic switchover.

The AIR properties are realized in Imago, a system for dependable, online upgrades in distributed systems. By installing the new version in a parallel universe (a distinct collection of resources), Imago isolates the production system from the upgrade operations and avoids breaking hidden dependencies. Imago relies on additional hardware and storage resources, which can be temporarily leased from an existing cloud-computing infrastructure. The end-to-end upgrade is an atomic operation, executed online even when performing complex schema and data con-versions, which commonly impose planned downtime. Moreover, I show that Imago reduces unplanned downtime, by conducting fault-injection experiments driven by my upgrade-centric fault model. These experiments outline a systematic approach for evaluating the dependability of online upgrades.

Relaxing the AIR properties opens the door to runtime behaviors that are poorly understood and difficult to ascertain. For example, the service-oriented architecture provides

weaker guarantees than the ISOLATION property because it does not address the interplay of change management (*e.g.* software upgrades) and distributed, autonomic management of service-level objectives. Similarly, in cases where the ATOMICITY property is difficult to enforce—for example, when upgrading distributed systems that span multiple administrative domains—the system is exposed to race conditions that involve multiple versions of the software. Such mixed-version races can induce critical inconsistencies, and they might be difficult to prevent using existing techniques. In consequence, performing software upgrades with relaxed AIR properties requires assessing the impact that the upgrade will have on the system.

One contribution of Imago is the separation of concerns between the functional aspects of an upgrade (*e.g.* converting persistent data) and the mechanisms for upgrading distributed systems online (*e.g.* switching atomically to the new version). This enables an upgrades-as-a-service model. In the future, upgrades-as-a-service will allow large-scale upgrades to be performed dependably and with fewer interventions from human operators.

10.2 Open questions and future work

The work presented in this dissertation raises a number of questions about the scientific foundations and the engineering choices that are required for providing dependable software upgrades. In general, systems that undergo runtime evolution (*e.g.*, online software-upgrades, architectural reconfigurations) must cope with changes implemented during the system's execution. These changes interact with the workload in ways that may be unpredictable at design-time. For example, dynamic software updates require programming techniques that may introduce new bugs [Hayden et al., 2009], transformations performed on persistent data may not be information-preserving [Curino et al., 2008a], distributed-system upgrades may break hidden dependencies [Dumitraș and Narasimhan, 2009a], and loading third-party components, at runtime, into plugin-based architectures can lead to unexpected behavior [Dumitraș et al., 2009].

Reasoning about such emerging behavior is difficult because previously-established system invariants do not hold, changes are implemented by both human and software agents, and externally-imposed deadlines might affect the outcome. Online-upgrade mech-

anisms are not acceptable for systems with strict certification requirements because, during the upgrade, the system behavior is not guaranteed to conform to the specification of either the old or the new version of the software [Segal, 2002].

This dissertation describes mechanisms that enable software-testing activities under operational conditions, in the deployment environment. Because online upgrades have unique failure modes, which cannot be fully analyzed before deployment, and because online upgrades provide few opportunities for testing the upgraded system, Runtime-testing will become an important property of future upgrading approaches. Moreover, this property can be implemented in a practical manner. While modern distributed systems can incorporate hundreds of thousands of nodes and can span multiple data centers, distributed worldwide, only a few of the environments in a production system are unique [Crameri et al., 2007]. In the parallel universe, Imago performs runtime testing in the environments and the scale at which the new version will operate; however, these future operating environments could be reproduced faithfully, but at a reduced scale, for testing purposes. In this case, the ingress interceptors would have to sample the live workload recorded because the system-under-test does not have the same capacity as the production system. Past research [for example: Oliveira et al., 2006; Crameri et al., 2007] has explored the failures that result from differences between the environments in which a system is tested and deployed. In the future, we should also investigate the impact of testing systems in environments that mirror faithfully all the attributes of the deployment environment—except for the scale targeted—in order to understand the limitations of Runtime-testing approaches.

In general, however, the presence of non-determinism or of behavioral changes prevents a direct comparison of the outputs of the old and new versions from producing meaningful results. While the question of what can be learned from the tests conducted at runtime remains open, this dissertation represents a first step toward a systematic approach for validating runtime-emerging behaviors.

For the next steps, we should turn our attention to robust mechanisms for incorporating online evolution into the design of distributed systems. While for the first 35 years of research on online software upgrades the focus has been on transparent upgrade mechanisms, which require limited or no cooperation from the system-under-upgrade, this proof of concept has largely been successful. We can execute complex changes that occur during

the evolution of real-world systems, with increased automation and with minimal downtime. This dissertation further discusses the principles of dependable software upgrades. In the future, we should focus on designing systems for upgrades, by trading transparency for an improved system dependability [Giuffrida and Tanenbaum, 2009].

Toward this goal, a promising topic for investigation is the design of programming languages with extensions for software upgrades. For example, the program changes that evolve into a new version of the software are usually buried in the revision-control system, across multiple branches and working copies. Language extensions could allow the programmer to check if various feature combinations are enabled in the current version of the code and to reason locally about the changes between versions. Currently, this can be achieved in practice with tangled **if**-ladders which allow an organization to decouple the deployment of new features from their activation, but which also render the code difficult to understand [Reiss, 2009]. Similarly, many recurring causes of planned downtime could be avoided by desiging a high-level language for specifying data transformations, from which the implementations of both the offline upgrade and the online upgrade of the database schema can be derived automatically [Downing, 2008]. Many of the ideas explored, in the programming-language community, for performing program refactorings and for providing refactoring support for software upgrades might apply for automating database-schema changes.

However, in order to assess whether such mechanisms are successful in improving the dependability of software upgrades—or to understand the reasons why they fail—we must be able to make quantitative comparisons of the availability and reliability of various upgrade mechanisms. The failures of software upgrades represent a sensitive subject, which prevents organizations from sharing the information required for replicating these failures outside of the deployment environments. To make progress in this direction, we must establish a comprehensive corpus of realistic faults that commonly occur during online upgrades, collected from multiple industry sources. Similar repositories, such as the top 25 programming errors that lead to security vulnerabilities [CWE/SANS, 2010], have had a significant impact on the practice of programming, and the qualitative benchmarking results presented in this dissertation emphasize the utility of an upgrade-centric fault repository.

Another topic that remains challenging is upgrading communication protocols. Because such upgrades usually require knowledge of the protocol semantics [Patel et al., 2003; Rütti et al., 2006; Anderson and Rathke, 2009], and generic protocol-upgrade mechanisms have not yet been developed. An alternative approach is to allow servers to submit the protocol stubs to the clients and to render the clients protocol-agnostic—as pioneered by the Jini middleware [Waldo, 2000]. In the past, this enabled Orbitz, an airline ticketing system and one of the early adopters of Jini technology, to perform seven major upgrades without failure and without downtime [Waldo, 2010].

This approach is currently being revisited, with the advent of cloud computing. Cloud-based distributed systems are able to send the appropriate client-side code to the users whenever they connect to the service (*e.g.*, through AJAX-style programming, but other forms of code migration might emerge in the future). This reduces the need for dynamic software updates on the client side (because the system is designed from the start to be able to load new code whenever needed) and for a large-scale dissemination of software updates (because the application logic is implemented on the server side and executes inside the service provider's data center). Moreover, thin clients are compelling for the providers because of the need to support resource-constrained mobile devices and because adding too much logic on the client side would lead to longer release cycles [Petrou, 2010]. This shift will bring into forefront a concern for correctly handling the dependencies among the components that might be loaded dynamically at the client-side and will emphasize the importance of the Isolation property.

The upgrade-as-a-service model represents another way of harnessing the opportunities provided by could computing for making online software upgrades easier to implement in practice. By separating the functional aspects of the upgrade (*e.g.* data conversions) from the non-functional mechanisms for online upgrade (*e.g.* atomic switchover), this model allows third-party providers to provide most of the infrastructure required for performing complex upgrades and renders online upgrades accessible to both large and small organizations. For example, the recent data-intensive applications, which mine large datasets in real time, use programming abstractions that continue to evolve. These applications will mandate competitive upgrades—replacing software components with alternative systems and converting the legacy data into a new format—that degrade the application's responsiveness by introducing planned downtime. In the future, upgrade-as-a-service could

minimize the downtime imposed by competitive upgrades. This approach could also assist companies in the migration to a virtualized or cloud-based infrastructure.

Appendices

Appendix A

NP-Completeness of the Package-Upgrade Problem

For Linux, the most common package formats are DEB [Silva, 2005], used by Debian and Ubuntu, and RPM [Bailey, 1997], used by Red Hat and Mandriva. Figure 1.1 shows the dependencies and conflicts among the DEB packages from a host running the Apache web server, with the PHP interpreter and the MySQL client library. For instance, the `mysql-client-5.0` package depends on 13 packages and it conflicts, owing to mutual incompatibilities, with one other package from the Debian repository. A package may also have alternative dependencies, *e.g.*, `apache2` has a dependency that can be satisfied by either one of three different packages. Dependency specifications often contain version constraints (*e.g.*, `>=4.1`) as well, and different versions of a package are mutually conflicting.[1]

The problem of upgrading[2] from a package p_{old} to a package p_{new}, which may or may not be a more recent version of p_{old}, entails finding a set of packages, including p_{new} but not p_{old}, such that all dependencies of the packages from this set are satisfied. Formally, a *repository* $R = (P, \hookrightarrow)$ contains a set P of *packages*, each package $p \in P$ having multiple versions $p_{v_1}, p_{v_2}, \ldots p_{v_k}$, and defines a *dependency relation* $\hookrightarrow$ between packages from P and combinations (conjunctions, disjunctions and negations) of packages:

- Package p' depends on another package p, which means that any version of p satisfies this dependency:
  $p' \hookrightarrow p \quad \equiv \quad p' \hookrightarrow p_{v_1} \vee \ldots \vee p_{v_k}$;

[1] Certain package management systems allow different versions of a package to be installed and to run side-by-side as long as they not not share configuration files or runtime state and they are isolated from each other [Leyden, 2003; Vermeulen et al., 2007].

[2] The proof follows the steps outlined by Di Cosmo [2005], while solving a different problem.

- Package p' depends on version v_i or greater of package p: $p' \hookrightarrow p_{v_i} \vee \ldots \vee p_{v_k}$ (version constraints using other operators than $\geq v_i$, while uncommon, are valid; they are handled similarly, by expanding all the available versions that satisfy the constraint);
- Package p' depends on either one of packages $p^{a_1}, p^{a_2}, \ldots p^{a_m}$ (alternative dependencies):[3] $p' \hookrightarrow p^{a_1} \vee p^{a_2} \vee \ldots \vee p^{a_m}$
- Package p' conflicts with package p: $p' \hookrightarrow \neg p$

A package may have several such dependencies, which have to be satisfied together. The general form of a dependency specification for a package p' is therefore an expression in the conjunctive normal form (CNF),[4] for example:

$$p' \hookrightarrow \underbrace{p^a}_{\text{one package}} \wedge \underbrace{(p^b_{v_1} \vee p^b_{v_2})}_{\text{version constraint}} \wedge \underbrace{(p^c \vee p^d \vee p^e)}_{\text{alternative packages}} \wedge \underbrace{\neg p^f}_{\text{conflict}} \wedge \ldots$$

A *configuration* α is a function $\alpha : R \mapsto \{0, 1\}$, which assigns to each package in R a boolean value indicating whether the package is installed or not. A *correct configuration* is a configuration α such that $\forall p$, $\alpha(p) = 1$ (all installed packages), the dependencies of p are satisfied in α. Given two packages p_{old} and p_{new} from a repository R and a correct configuration α_{old}, with $\alpha_{old}(p_{old}) = 1$ and $\alpha_{old}(p_{new}) = 0$, the upgrade problem is formally defined as follows:

$$Upgrade(\alpha_{old}, p_{old}, p_{new}, R) = \{\exists \text{ correct configuration } \alpha_{new} : \alpha_{new}(p_{old}) = 0 \text{ and } \alpha_{new}(p_{new}) = 1\}$$

Theorem 1 *$Upgrade(\alpha_{old}, p_{old}, p_{new}, R)$ is NP-complete.*

Proof. Firstly, I show that a solution α_{new} can be verified to be correct through an algorithm that is polynomial in the size of R (*i.e.*, $Upgrade(\alpha_{old}, p_{old}, p_{new}, R)$ is NP). Checking that $\alpha_{new}(p_{old}) = 0$ and $\alpha_{new}(p_{new}) = 1$ has a constant complexity $O(1)$. Then, $\forall p \in R$ such that $\alpha_{new}(p) = 1$: check whether their CNF dependency specification of p is also satisfied in α_{new}. This step requires evaluating a subset of all the clauses stored in R and is therefore linear in the size of R.

[3] Alternative dependencies may be expressed either directly, using the "|" operator (in DEB packages), or through "virtual packages," which are names corresponding to functionality that may provided by several regular packages (for both DEB and RPM packages).

[4] The dependency specifications of both DEB and RPM package formats can be formulated as a CNF boolean expression [Mancinelli et al., 2006].

Secondly, I present a polynomial reduction from an instance of a known NP-complete problem, 3SAT [Garey and Johnson, 1979], to the package-upgrade problem (*i.e.*, $Upgrade(\alpha_{old}, p_{old}, p_{new}, R)$ is NP-hard). Let $S = C_1 \wedge \ldots \wedge C_n$ be an instance of 3SAT, with each C_i a disjunction of three literals $C_i = l_{i,1} \vee l_{i,2} \vee l_{i,3}$. Each literal $l_{i,j}$ is a boolean variable x or its negation $\neg x$. Let $X = \{x_1, \ldots x_k\}$ be the set of boolean variables occurring in S.

The reduction algorithm defines a configuration α_{old} and a repository R_S containing: two packages p_{old} and p_{new}, such that $\alpha_{old}(p_{old}) = 1$ and $\forall p \in R_S, p \neq p_{old} : \alpha_{old}(p) = 0$; one package p_{C_i} for each clause C_i; and packages p^a_x, p_x and $p_{\neg x}$ for each variable x. These packages have the following dependencies (the complexity of each transformation is specified on the right side):

1. p_{old} does not have any dependencies and can be installed without any further checks
2. p_{new}, which corresponds to S, depends on $p_{C_1}, \ldots p_{C_n}, p^a_{x_1}, \ldots p^a_{x_k}$:
   $p_{new} \hookrightarrow p_{C_1} \wedge \ldots \wedge p_{C_n} \wedge p^a_{x_1} \wedge \ldots \wedge p^a_{x_k}$ $O(n+k)$
3. For each clause C_i, p_{C_i} has alternative dependencies $p_{l_{i,1}}$, $p_{l_{i,2}}$ and $p_{l_{i,3}}$:
   $p_{C_i} \hookrightarrow p_{l_{i,1}} \vee p_{l_{i,2}} \vee p_{l_{i,3}}$ $O(n)$
4. For each variable x, p^a_x has alternative dependencies p_a and $p_{\neg a}$:
   $p^a_x \hookrightarrow p_x \vee p_{\neg x}$ $O(k)$
5. For each variable x, p_x conflicts $p_{\neg x}$ and $p_{\neg x}$ conflicts p_x:
   $p_x \hookrightarrow \neg p_{\neg x}, p_{\neg x} \hookrightarrow \neg p_x$ $O(k)$

This transformation algorithm is linear in the size of the problem S; its complexity is the complexity of the second step, $O(n+k)$, where n is the number of clauses from S and k is the number of boolean variables occurring in S.

If there is a solution f to the problem S, then define a configuration α_{new} as follows: $\alpha_{new}(p_{new}) = \alpha(p_{C_1}) = \ldots = \alpha(p_{C_n}) = 1$, $\alpha_{new}(p_{old}) = 0$ and, for each variable x, $\alpha_{new}(P_x) = f(x)$ and $\alpha_{new}(p_{\neg x}) = f(\neg x)$. Because configuration α_{new} contains p_{new} but not p_{old} and because constraints 1–5 are satisfied for all the installed packages in α_{new}, p_{old} can be upgraded to p_{new} and α_{new} is a solution to the problem $Upgrade(\alpha_{old}, p_{old}, p_{new}, R_S)$.

Conversely, if α_{new} is a solution to the problem $Upgrade(\alpha_{old}, p_{old}, p_{new}, R_S)$, then constraint 2 is satisfied and $\forall i, 1 \leq i \leq n : \alpha_{new}(p_{C_i}) = 1$ and $\forall x \in X, \alpha_{new}(p^a_x) = 1$. Constraint 5 implies that, for each variable x, either p_x or $p_{\neg x}$ is installed. Constraint 3 implies that, for each clause C_i, at least one of $p_{l_{i,1}}$, $p_{l_{i,2}}$ and $p_{l_{i,3}}$ is installed. Define a valuation f as $f(x) = 1$ if $\alpha_{new}(p_x) = 1$ and $f(a) = 0$ otherwise. f satisfies S because

all the disjunctive clauses $C_1, \ldots C_n$ are true. This proves that the transformation from S to $Upgrade(\alpha_{old}, p_{old}, p_{new}, R_S)$ is a polynomial reduction and, hence, that the package-upgrade problem is NP-hard.

As it is both NP and NP-hard, the package-upgrade problem is NP-complete. In fact, the problem of installing a package in an empty system (*i.e.*, which does not contain any installed packages) is also NP-complete [Di Cosmo, 2005].

Appendix B

List of Upgrade Faults

Fault name	Source	Page/ Bug Id	Description	Location	Root cause	Root cause (subclass)	Hidden dependency	Original classification	Cognitive level	Reported effect	Cluster
apache_42627	Field study	42627	Unable to authenticate using authz-ldap require group, due to escaped characters in group name	Middle tier	Configuration	Typo	none	Syntax error	skill	Incorrect functionality	1
apache_config_noapp	User study	4	Added info about the new app server, but forgot to add the machine's name to the line that specifies app-server names	Front-end	Configuration	Structural	network address	Global misconfiguration	rule	Latent error (frontend cannot contact app server)	1
apache_config_nochange	User study	5	Forgot to modify the Apache configuration file altogether	Front-end	Configuration	Structural	none	Global misconfiguration	rule	Throughput degradation	1
apache_config_nomount	User study	6	Forgot to specify how Apache should map a given URL to a request for a Tomcat servlet	Front-end	Configuration	Structural	network address	Global misconfiguration	rule	Throughput degradation (affected web server unable to process client requests)	1
apache_config_staticpath	User study	6	Configured new server to get static files / heartbeat program from the old Apache's directory tree	Front-end	Configuration	Structural	file path	Local misconfiguration	rule	Latent error (when old distribution removed; new server cannot serve static files)	1
apache_config_staticpath	Field study	43111	/htdocs getting served when not configured	Front-end	Configuration	Structural	file path	Paths and permissions	rule	Error message	1
apache_config_typo	User study	4	Syntax error in Apache configuration file	Front-end	Configuration	Typo	none	Local misconfiguration	skill	Throughput degradation (mod_jk crashed)	1
apache_config_typo	Field study	42138	Syntax error in httpd.conf	Front-end	Configuration	Typo	none	Syntax error	skill	Error message	1
apache_config_wrongpath	User study	6	Incorrectly specified the path to the heartbeat program	Front-end	Configuration	Typo	file path	Global misconfiguration	skill	Throughput degradation (mod_jk crashed)	1
wrong_apache	User study	5	Reconfigured one Apache distribution and launched the executable from another	Front-end	Procedure		file path	Start of wrong SW version	skill	Throughput degradation (affected web server unable to process client requests)	1
wrong_apache	User study	5	Launched Apache from the wrong distribution while restarting the service	Front-end	Procedure		file path	Start of wrong SW version	skill	Service inaccessible (both frontend servers unable to process client requests)	1
wrong_shutdown_frontend	User study	5	Shutdown both frontend web-servers at the same time	Front-end	Procedure	Order inversion	replication degree	Global misconfiguration	skill	Throughput degradation (both frontend servers inaccessible during restart)	1
apache_41751	Field study	41751	Keepalive connections keep children tied up even if new requests starve		Configuration	Semantic	request scheduling	Parameter tuning	knowledge	Performance degradation	2
apache_41920	Field study	41920	error msg: Invalid command 'Order' after upgrading to httpd-2.2.4	Middle tier	Configuration	Semantic	parameter interdependency	Semantic error	knowledge	Error message	2
apache_41979	Field study	41979	Load Balancer Manager Web Client Is Blank due to configuration error	Front-end	Configuration	Semantic	none	Semantic error	knowledge	Incorrect functionality	2
apache_43395	Field study	43395	RewriteRule and Location or Directory does not work, due to syntax error	Front-end	Configuration	Semantic	parameter interdependency	Semantic error	knowledge	Error message	2
apache_config_samename	User study	4	Configured identical names for the application servers in the Tomcat connector	Front-end	Configuration	Semantic	parameter interdependency	Local misconfiguration	knowledge	Service inaccessible (mod_jk crashed on both frontend servers)	2
apache_largefile	Field study	42751	CIFS mounted filesystems do not transmit files	Middle tier	Configuration	Semantic	cached data	Parameter tuning	knowledge	Incorrect functionality	2
apache_largefile	Field study	43232	error while transmitting file over 64K	Middle tier	Configuration	Semantic	cached data	Parameter tuning	knowledge	Incorrect functionality	2
apache_largefile	Field study	42340	Client serving fails when accessing network mounted device.	Middle tier	Configuration	Semantic	cached data	Parameter tuning	knowledge	Incorrect functionality	2
apache_satisfy	Field study	42709	.htaccess is viewable by browser after login validation, due to incorrect Satisfy directive	Middle tier	Configuration	Semantic	parameter interdependency	Semantic error	knowledge	Incorrect functionality	2
apache_servername	Field study	42605	HTTPS-URL with special-port requesting a directory (without trailing slash) is rewritten to HTTPS-URL requesting a directory with trailing slash, BUT the specified port is missing	Front-end	Configuration	Semantic	none	Parameter tuning	knowledge	Incorrect functionality	2

Unique name	Source	Page/ Bug Id	Description	Location	Root cause	Root cause (subclass)	Hidden dependency	Original classification	Cognitive level	Reported effect	Cluster
apache_41358	Field study	41358	No DSO works: ./configure --enable-mods-shared doesn't work, neither does apxs, neither does ./configure --enable-proxy=shared.	Front-end	Procedure		dynamic linking	Build	rule	Error message	3
apache_41617	Field study	41617	undefined reference to `ap_cache_generate_name'		Procedure		none	Build	rule	Error message	3
apache_42332	Field study	42332	GDBM not supported in 2.2.4?	Middle tier	Procedure		dynamic linking	Build	rule	Error message	3
apache_42975	Field study	42975	ProxyPassReverse not handling relative redirects properly	Front-end	Procedure		communication protocol	Environmental conflict	rule	Incorrect functionality	3
apache_43328	Field study	43328	mod_authnz_ldap does not compile	Middle tier	Configuration		dynamic linking	Build	rule	Error message	3
apache_43518	Field study	43518	no listening sockets available		Procedure	Omission	listening port	Environmental conflict	rule	Error message	3
apache_43523	Field study	43523	Compatibility with last svn dav and authz .so modules	Middle tier	Procedure		dynamic linking	Build	rule	Error message	3
apache_43945	Field study	43945	httpd is hanging because it waits for random data from /dev/random. This device blocks if not enough entropy is present		Procedure		entropy available	Build	rule	Hang	3
apache_43986	Field study	43986	Multiple compile errors in mod_disk_cache.c		Procedure		none	Build	rule	Error message	3
apache_libconflict	Field study	42285	mod_authnz_ldap reports [Can't contact LDAP server]	Middle tier	Procedure		dynamic linking	Environmental conflict	rule	Error message	3
apache_libconflict	Field study	42556	Apache with LDAP support segfaults on Solaris 9 with LDAP	Middle tier	Procedure		dynamic linking	Environmental conflict	rule	Crash	3
incorrect_tomcat_noroot	User study	5	Tomcat started incorrectly (forgot to obtain root privileges before starting) on app server	Middle tier	Procedure	Omission	access privileges	Incorrect restart	rule	Throughput degradation (Tomcat silently died on new app server)	3
db_deployment_script	Survey	5	Bugs in the scripts for deploying DB changes	Back-end	Procedure			Deployment problems			4
db_performance_tuning	Survey	6	Erroneous performance tuning	Back-end	Configuration			Performance problems	knowledge	Poor performance to applications or users	4
db_schema_design	Survey	6	Inappropriate database-structure design	Back-end	Procedure		schema design	Performance problems	knowledge	Poor performance to applications or users	4
db_schema_design	Survey	6	Incorrect database design (e.g. duplicated identity columns; columns too small to hold data)	Back-end	Procedure		schema design	General structure problems	knowledge	Deadlocks; etc.	4
db_schema_mismatch	Survey	5	Forgot to change the schema of the online DB before deploying a new or changed application	Back-end	Procedure	Wrong action	schema design	Deployment problems	knowledge	Fatal SQL errors	4
db_schema_mismatch	Survey	6	Applications compiled agains the wrong schema are deployed online	Back-end	Procedure	Omission	schema design	Deployment problems	knowledge	Fatal SQL errors	4
dbms_misconfiguration	Survey	7	Misconfigured DBMS	Back-end	Configuration			General maintenance problems	knowledge	Could not restart DBMS	4
deploy_accidental_changes	Survey	6	Accidentally propagated the changes made to the database in the testing environment	Back-end	Procedure	Wrong action	component online	Deployment problems	rule		4
deploy_index_nochange	Survey	6	Forgot to reapply indexes in the production database	Back-end	Procedure	Omission	schema design	Deployment problems	rule		4
deploy_wrong_changes	Survey	6	Inappropriate changes directly to the online database	Back-end	Procedure	Wrong action	component online	Deployment problems	rule		4
mysql_nopass_root	User study	5	No password set up for MySQL root user	Back-end	Configuration		access privileges	Local misconfiguration	rule	Security vulnerability	4
no_replication	Survey	7	Forgot to restart the DBMS replication	Back-end	Procedure	Omission	replication degree	General maintenance problems	rule		4
no_space	Survey	6	Disk space exhaustion	Back-end	Configuration		storage space	Space problems	rule		4
no_space_backup	Survey	7	Insufficient backup space	Back-end	Procedure	Omission	storage space	General maintenance problems	rule		4
tablespace_full	Survey	6	Tablespace problems	Back-end	Configuration		storage space	Space problems	rule		4
wrong_db_disk	User study	5	Installed the DB on a slow disk	Back-end	Procedure	Wrong action	disk speed	Wrong choice of HW component	rule	Throughput degradation (limited capacity)	4
wrong_index	Survey	6	Inappropriate indexing scheme	Back-end	Procedure		schema design	Performance problems	knowledge	Poor performance to applications or users	4
wrong_privileges_excessive	Survey	6	Excessive access privileges granted to some users or applications	Back-end	Configuration		access privileges	Access-privilege problems	rule	Security vulnerability	4
wrong_privileges_insufficient	User study	5	MySQL user not given necessary privileges	Back-end	Configuration		access privileges	Global misconfiguration	rule	Service inaccessible (all Tomcat threads blocked)	4
wrong_privileges_insufficient	Survey	6	Insufficient acccess privileges granted to users and applications	Back-end	Configuration		access privileges	Access-privilege problems	rule	Inability to access the whole or parts of the) database	4
wrong_shutdown_db	Survey	7	Incorrectly shut down the database	Back-end	Procedure	Spurious action	replication degree	General maintenance problems	skill	Database inaccessible to the application servers	4

Appendix C

Upgrade Risk Model: Implementation

To ensure the reproducibility of the results reported in Chapter 9, I include here the Java program that implements the upgrade risk model. This program was used for determining whether to upgrade or not to upgrade in the case sudies from Section 9.3.

```
/*****************************************************************************
 * UpgradeRiskModel.java  ---
 *
 *   Analytical methods for evaluating the risk of a rolling upgrade.
 *
 *   (c) Tudor Dumitras, 2010
 *
 *   $Id: UpgradeRiskModel.java 71 2010-10-21 23:43:12Z tudor $
 *****************************************************************************/

import java.util.Random;
import java.util.Arrays;


public class UpgradeRiskModel
{
    // Parameters of the upgrade risk
    private long    U;                  // Number of frontends to upgrade
    private double  tau_mean;           // Mean time-to-upgrade for 1 host
    private double  tau_lo;             // Lower bound of the time-to-upgrade
    private double  tau_hi;             // Upper bound of the time-to-upgrade
    private long    C_new;              // # callbacks invoked by the new version
    private int     S_max;              // Maximum severity level
    private int     N_call;             // Number of potential inconsistencies
    private int     p_call[];           //   Conditional probability of
                                        //   triggering an inconsistency, given
                                        //   that a new callback arrives at the
                                        //   old version (N_call elements).
                                        //   This is a discrete probability spec:
                                        //   p_lo = 1, p_med = 2, p_hi = 3.
```

```
    private int     S_inc[];          //   Severity of the inconsistency
                                      //   (N_call elements).

    // Parameters of the "no upgrade" risk
    private int     N_bug;            // Number of bugs fixed by the upgrade
    private int     p_bug[];          //   Probability of hitting a bug
                                      //   (N_bug elements). This is a discrete
                                      //   probability spec:
                                      //   p_lo = 1, p_med = 2, p_hi = 3.
    private int     S_bug[];          //   Severity of the bug (N_bug elements)

    // Other local fields
    private Random  rnd_gen;          // random number generator
    private double  cdf_mode;         // CDF of tau_mean
    private static int sim_iterations = 10000;  // # of Monte Carlo iterations
    private static double z = 1.96;  // z-score for Monte Carlo confid. interv.
                                     // - for 90%    z = 1.645
                                     // - for 95%    z = 1.96
                                     // - for 99%    z = 2.575


    // Constructor
    public UpgradeRiskModel(long    U,
                            double  tau_mean,
                            double  tau_lo,
                            double  tau_hi,
                            long    C_new,
                            int     S_max,
                            int     N_call,
                            int     p_call[],
                            int     S_inc[],
                            int     N_bug,
                            int     p_bug[],
                            int     S_bug[])
    {
        this.U        = U;
        this.tau_mean = tau_mean;
        this.tau_lo   = tau_lo;
        this.tau_hi   = tau_hi;
        this.C_new    = C_new;
        this.S_max    = S_max;
        this.N_call   = N_call;
        this.p_call   = Arrays.copyOf (p_call, N_call);
        this.S_inc    = Arrays.copyOf (S_inc, N_call);

        this.N_bug    = N_bug;
        this.p_bug    = Arrays.copyOf (p_bug, N_bug);
        this.S_bug    = Arrays.copyOf (S_bug, N_bug);

        // Seed the random number generator using the current time
        rnd_gen = new Random();
```

```
        // Compute the CDF at the mode of the triangle distribution
        cdf_mode = (double)(tau_mean - tau_lo) / (tau_hi - tau_lo);
    }

    // Returns the risk of not upgrading
    public double getRiskNoUpgrade()
    {
        double result = 0.0;

        for (int i=0; i<N_bug; i++)
            result += (p_bug[i] * S_bug[i]);

        return result / N_bug / S_max;
    }


    // Computes the risk of not upgrading, without taking into account the
    // time-variable probability component
    private double getFixedRiskUpgrade()
    {
        double result = 0.0;

        for (int i=0; i<N_call; i++)
            result += (p_call[i] * S_inc[i]);

        return result / N_call / S_max;
    }

    // Returns the maximum risk of upgrading during an upgrade
    public double maxRiskUpgrade()
    {
        double clbk = C_new;
        double max_P_bad_callback = Math.pow(1/(clbk + 1),
                                             1/clbk) * clbk / (clbk + 1);

        return max_P_bad_callback * getFixedRiskUpgrade();
    }


    // Estimates the mean probability of inconsistency through a Monte Carlo
    // simulation with a fixed number of iterations. Returns an array with
    // three values:
    // - the probability estimate (i.e. the mean of the simulation runs)
    // - the low bound of the 95% confidence interval
    // - the high bound of the 95% confidence interval
    public double[] meanRiskUpgradeMonteCarlo ()
    {
        double results[] = new double[3];
        double simulationResults[] = new double[sim_iterations];
        double sum, sum_err, sd;
```

```
        // Run simulation
        int i=0, actual_iterations = 0;;
        for (//this loop could be parallelized, for better performance
             int iter=0; iter<sim_iterations; iter++) {

            double result = meanSimulationIteration();

            if (result > 0)       // Avoid NaNs
                simulationResults[i++] = result;
        }

        // Record the number of successful simulation runs
        actual_iterations = i;

        // Compute the mean of the simulation runs
        sum = 0.0;
        for (int iter=0; iter<actual_iterations; iter++) {
            sum += simulationResults[iter];
        }

        results[0] = sum / (double)actual_iterations;

        // Compute the sample standard deviation
        sum_err = 0.0;
        for (int iter=0; iter<actual_iterations; iter++) {
            double err = simulationResults[iter] - results[0];
            sum_err += err * err;
        }

        sd = Math.sqrt(sum_err / (double)(actual_iterations - 1));

        // Compute the confidence interval
        results[1] = results[0] /*mean*/ -
            z /*z-score*/ * sd / Math.sqrt((double)U) /*std. error*/;
        results[2] = results[0] /*mean*/ +
            z /*z-score*/ * sd / Math.sqrt((double)U) /*std. error*/;

        // Incorporate the conditional probability of callback errors and
        // the severity of the incompatibility
        results[0] *= getFixedRiskUpgrade();
        results[1] *= getFixedRiskUpgrade();
        results[2] *= getFixedRiskUpgrade();

        return results;
    }

    private double meanSimulationIteration ()
    {
        double sum_ptime = 0;
        double sum_time  = 0;
```

```
        for (int j=0; j<U; j++) {
            double time, probability;

            // Generate the time needed to upgrade server j
            time = nextTriangle();

            // Compute the probability of inconsistency
            probability = (double)j / U * (1 - Math.pow((double)j / U,
                                                        C_new));

            // Update accumulators
            sum_time += time;
            sum_ptime += (time * probability);
        }

        // Compute the mean probability of inconsistency
        return sum_ptime / sum_time;
    }


    // Returns a random number drawn from a triangle distribution, with mean
    // mean_tau and bounds [tau_lo, tau_hi]. Ensures that the distribution
    // has a positive support (i.e., no negative numbers).
    private double nextTriangle ()
    {
        double result;
        double uniform = rnd_gen.nextDouble();

        result = uniform < cdf_mode ?
            tau_lo + Math.sqrt(uniform *
                               (tau_hi - tau_lo) * (tau_mean - tau_lo)) :
            tau_hi - Math.sqrt((1 - uniform) *
                               (tau_hi - tau_lo) * (tau_hi - tau_mean));

        return result > 0 ? result : 0;
    }

    // Returns a string (low, medium or high) that corresponds to the
    // discrete risk level.
    public String riskLevel (double risk)
    {
        if (risk > 2.0) return "high";
        if (risk > 1.0) return "medium";
        return "low";
    }
}
```

Bibliography

S. Ajmani, B. Liskov, and L. Shrira, "Modular software upgrades for distributed systems," in *European Conference on Object-Oriented Programming*, Nantes, France, Jul 2006, pp. 452–476. 8, 20, 26, 50, 69, 70, 93, 106, 129

Y. Amir, C. Danilov, and J. Stanton, "A low latency, loss tolerant architecture and protocol for wide area group communication," in *International Conference on Dependable Systems and Networks*, New York, NY, Jun 2000, pp. 327–336. 81

C. Amza *et al.*, "Specification and implementation of dynamic web site benchmarks," in *IEEE Workshop on Workload Characterization*, Austin, TX, Nov 2002, pp. 3–13, http://rubis.objectweb.org/. 94

A. Anderson and J. Rathke, "Migrating protocols in multi-threaded message-passing systems," in *ACM Workshop on Hot Topics in Software Upgrades*, Orlando, FL, Oct 2009. 153

E. Anderson, M. Hobbs, K. Keeton, S. Spence, M. Uysal, and A. Veitch, "Hippodrome: Running circles around storage administration," in *USENIX Conference on File and Storage Technologies (FAST '02)*, Monterey, CA, January 2002, p. 13. 30, 112, 121, 124

R. Anderson, "The end of DLL Hell," *MSDN Magazine*, Jan 2000. [Online]. Available: http://msdn2.microsoft.com/en-us/library/ms811694.aspx 5, 18

M. Arlitt and T. Jin, "A workload characterization study of the 1998 World Cup Web site," *IEEE Network*, vol. 14, no. 3, pp. 30–37, May 2000. 111

M. F. Arlitt and C. L. Williamson, "Web server workload characterization: the search for invariants," *SIGMETRICS Performance Evaluation Review*, vol. 24, no. 1, pp. 126–137, 1996. 111

J. Armstrong, R. Virding, C. Wikstrom, and M. Williams, *Concurrent programming in Erlang*, 2nd ed. Prentice Hall, 1996. 24

J. Arnold and F. Kaashoek, "Ksplice: automatic rebootless kernel updates," in *ACM European Conference on Computer Systems*, 2009, pp. 187–198. 23, 92

A. Avižienis, J.-C. Laprie, B. Randell, and C. Landwehr, "Basic concepts and taxonomy of dependable and secure computing," *IEEE Transactions on Dependable and Secure Computing*, vol. 1, no. 1, pp. 11 – 33, Jan 2004. 1, 3, 96

E. Bailey, *Maximum RPM*. RedHat Press, Indianapolis, IN, 1997. 156

H. Baltazar, "Database database replication is the ticket," eWEEK, Feb 2005. [Online]. Available: http://www.eweek.com/c/a/Database/Database-Replication-Is-the-Ticket/ 62

A. Baumann, J. Appavoo, R. W. Wisniewski, D. Da Silva, O. Krieger, and G. Heiser, "Reboots are for hardware: challenges and solutions to updating an operating system on the fly," in *USENIX Annual Technical Conference*, 2007, pp. 1–14. 23

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*, Philadelphia, PA, Nov 2002, pp. 233–242. 29

R. C. Beatty and C. D. Williams, "ERP II: best practices for successfully implementing an ERP upgrade," *Communication of the ACM*, vol. 49, no. 3, pp. 105–109, Mar 2006. 50, 87

H. Berenson, P. Bernstein, J. Gray, J. Melton, E. O'Neil, and P. O'Neil, "A critique of ANSI SQL isolation levels," in *ACM SIGMOD International Conference on Management of Data*, San Jose, CA, May 1995, pp. 1–10. 71

P. A. Bernstein and L. M. Haas, "Information integration in the enterprise," *Communications of the ACM*, vol. 51, no. 9, pp. 72–79, Sep 2008. 8, 51

P. Bhattacharya and I. Neamtiu, "Dynamic updates for Web and cloud applications," in *Workshop on Analysis and Programming Languages for Web Applications and Cloud Applications*, Toronto, Canada, 2010, pp. 21–25. 23, 26, 70, 113

G. Bierman, M. Parkinson, and J. Noble, "UpgradeJ: Incremental typechecking for class upgrades," in *European Conference on Object-Oriented Programming*, Paphos, Cypress, Jul 2008, pp. 235–259. 24

T. Bloom, "Dynamic module replacement in a distributed programming system," Ph.D. dissertation, MIT, 1983. 8, 25, 69, 70

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. 146

C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman, "Lazy modular upgrades in persistent object stores," in *Object-Oriented Programing, Systems, Languages and Applications*, Anaheim, CA, Oct 2003, pp. 403–417. 7, 19, 24

E. A. Brewer, "Lessons from giant-scale services," *IEEE Internet Computing*, vol. 5, no. 4, pp. 46–55, Jul/Aug 2001. 8, 28, 48, 54, 55, 62, 89, 92, 96

F. P. Brooks, Jr., "No silver bullet: Essence and accidents of software engineering," *Computer*, vol. 20, no. 4, pp. 10–19, 1987. 1

A. Brown, G. Kar, and A. Keller, "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment," in *Integrated Network Management*, Seattle, WA, May 2001, pp. 377–390. 6, 23

R. Brown and J. Pickard, *Yum (Yellowdog Updater, Modified) HOWTO*, Sep 2003. [Online]. Available: http://www.phy.duke.edu/~rgb/General/yum_HOWTO/yum_HOWTO 5, 7, 21, 22

G. J. Buban, P. V. Donlan, A. Marinescu, T. D. McGuire, D. B. Probert, H. H. Vo, and Z. Wang, "Patching of in-use functions on a running computer system," Mar 2004, US Patent Application 20040107416, assigned to Microsoft Corporation. 23

G. Candea, S. Kawamoto, Y. Fujiki, G. Friedman, and A. Fox, "Microreboot – a technique for cheap recovery," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004, pp. 31–44. 94

N. Carvalho, A. C. Jr., J. Pereira, L. Rodrigues, R. Oliveira, and S. Guedes, "On the use of a reflective architecture to augment database management systems," *Journal of Universal Computer Science*, vol. 13, no. 8, pp. 1110–1135, 2007. [Online]. Available: http://www.jucs.org/jucs˙13˙8/on˙the˙use˙of 79

E. Cecchet, J. Marguerite, and W. Zwaenepoel, "C-JDBC: Flexible database clustering middleware," in *USENIX Annual Technical Conference*, 2004. 79, 85

C. Chatfield, *Statistics for Technology: A Course in Applied Statistics*, 3rd ed. Chapman & Hall/CRC, 1983. 45, 103

D. Chess, G. Pacifici, and A. Tantawi, "Experience with collaborating managers: Node group manager and provisioning manager," in *International Conference on Automatic Computing*, Seattle, WA, Jun 2005, pp. 39–50. 112, 113, 118

A. Choi, "Online application upgrade using edition-based redefinition," in *ACM Workshop on Hot Topics in Software Upgrades*, Orlando, FL, Oct 2009. 2, 29, 62, 66, 131

B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, and R. Sears, "Benchmarking cloud serving systems with YCSB," in *ACM Symposium on Cloud Computing*, Indianapolis, IN, 2010, pp. 143–154. 94

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*, Stevenson, WA, Oct 2007, pp. 221–236. 2, 7, 17, 26, 29, 32, 33, 35, 45, 47, 130, 151

A. Cruz, "Update on today's Gmail outage," The Official Gmail Blog, Feb 2009. [Online]. Available: http://gmailblog.blogspot.com/2009/02/update-on-todays-gmail-outage.html 4

C. A. Curino, H. J. Moon, L. Tanca, and C. Zaniolo, "Schema evolution in Wikipedia: toward a Web information system benchmark," in *International Conference on Enterprise Information Systems*, Barcelona, Spain, Jun 2008b. 18, 19, 53, 58, 61, 75

C. A. Curino, H. J. Moon, and C. Zaniolo, "Graceful database schema evolution: the PRISM workbench," in *International Conference on Very Large Data Bases (VLDB)*, Auckland, New Zealand, Aug 2008a. 8, 20, 51, 91, 98, 150

CWE/SANS, "Top 25 most dangerous programming errors," Feb 2010. [Online]. Available: http://cwe.mitre.org/top25/ 143, 152

J. Dean and S. Ghemawat, "MapReduce: Simplified data processing on large clusters," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004, pp. 137–150. 86

D. J. DeWitt, "The Wisconsin benchmark: Past, present, and future," in *The Benchmark Handbook for Database and Transaction Systems*, J. Gray, Ed. Morgan Kaufmann, 1993. 90, 94

R. Di Cosmo, "Report on formal management of software dependencies," INRIA, Tech. Rep., Sep 2005, (EDOS Project Deliverable WP2-D2.1). 7, 22, 156, 159

R. Di Cosmo, S. Zacchiroli, and P. Trezentos, "Package upgrades in FOSS distributions: details and challenges," in *ACM Workshop on Hot Topics in Software Upgrades*, Oct 2008. 7, 21, 22, 48

D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance and Evolution (JSME)*, vol. 18, no. 2, pp. 83–107, Mar/Apr 2006. [Online]. Available: http://dx.doi.org/10.1002/smr.328 5, 18, 19, 22, 32

D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated detection of refactorings in evolving components," in *European Conference on Object-Oriented Programming*, Nantes, France, Jul 2006, pp. 404–428. [Online]. Available: http://netfiles.uiuc.edu/dig/papers/DetectRefactoringsECOOP.pdf 5, 22, 47, 93

X. Ding, H. Huang, Y. Ruan, A. Shaikh, B. Peterson, and X. Zhang, "Splitter: A proxy-based approach for post-migration testing of Web applications," in *European Conference on Computer Systems*, Paris, France, Apr 2010, pp. 97–110. 27, 28, 85, 93

E. Dolstra and A. Löh, "NixOS: A purely functional Linux distribution," in *ACM SIGPLAN International Conference on Functional programming*, Victoria, BC, Canada, 2008, pp. 367–378. 21

A. Downing (Oracle Corporation), Personal communication, 2008. 28, 48, 51, 59, 87, 100, 131, 152

T. Dumitraș and P. Narasimhan, "No downtime for data conversions: Rethinking hot upgrades," Carnegie Mellon University, Tech. Rep. CMU-PDL-09-106, 2009b. 52

T. Dumitraș and P. Narasimhan, "Toward upgrades-as-a-service in distributed systems," in *ACM/IEEE/IFIP Middleware Conference*, Urbana-Champaign, IL, Nov/Dec 2009c, poster. 84

T. Dumitraș 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*, Urbana-Champaign, IL, Nov/Dec 2009a, pp. 349–372. 37, 49, 70, 150

T. Dumitraș, D. Roșu, A. Dan, and P. Narasimhan, "Ecotopia: An ecological framework for change management in distributed systems," in *Architecting Dependable Systems IV*, C. Gacek, A. Romanovsky, and R. de Lemos, Eds. Springer-Verlag, LNCS 4615, 2007b, pp. 262–286. 113

T. Dumitraș, J. Tan, Z. Gho, and P. Narasimhan, "No more HotDependencies: Toward dependency-agnostic upgrades in distributed systems," in *Workshop on Hot Topics in System Dependability*, Edinburgh, Scotland, Jun 2007a. 68

T. Dumitraș, S. Kavulya, and P. Narasimhan, "A fault model for upgrades in distributed systems," Carnegie Mellon University, Tech. Rep. CMU-PDL-08-115, 2008. 35, 36, 39, 102

T. Dumitraș, F. Eliassen, K. Geihs, H. Muccini, A. Polini, and T. Ungerer, "Testing runtime evolving systems," in *Self-Healing and Self-Adaptive Systems*, ser. Dagstuhl Seminar Proceedings, Dagstuhl, Germany, 2009. 150

T. Dumitraș, E. Tilevich, and P. Narasimhan, "To upgrade or not to upgrade: Impact of online upgrades across multiple administrative domains," in *ACM SPLASH Onward!*, Reno/Tahoe, NV, Oct 2010. 127

J. Dunagan, R. Roussev, B. Daniels, A. Johson, C. Verbowski, and Y.-M. Wang, "Towards a self-managing software patching process using black-box persistent-state manifests," in *International Conference on Autonomic Computing*, New York, NY, May 2004, pp. 106–113. 6, 23

A. Egyed, "A scenario-driven approach to trace dependency analysis," *IEEE Transactions on Software Engineering*, vol. 29, no. 2, pp. 116–132, 2003. 5

R. Fabry, "How to design a system in which modules can be changed on the fly," in *International Conference on Software Engineering*, San Francisco, CA, 1976, pp. 470–476. 19, 23, 88

R. Fagin, "Inverting schema mappings," *ACM Transactions on Database Systems*, vol. 32, no. 4, 2007. 20

F. Ferrandina, T. Meyer, R. Zicari, G. Ferran, and J. Madec, "Schema and database evolution in the O2 object database system," in *International Conference on Very Large Data Bases*, Zürich, Switzerland, Sep 1995, pp. 170–181. 8, 51

M. Fowler, "Inversion of Control Containers and the Dependency Injection pattern," Jan 2004. [Online]. Available: http://martinfowler.com/articles/injection.html 6

M. Garey and D. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*. W. H. Freeman, 1979. 158

C. Giuffrida and A. S. Tanenbaum, "Cooperative update: a new model for dependable live update," in *ACM Workshop on Hot Topics in Software Upgrades*, Orlando, FL, Oct 2009. 152

Global Grid Forum, "Web services agreement specification (WS-Agreement)," Aug 2004, draft, version 11. 115

R. A. Golding and T. M. Wong, "Walking toward moving goalposts: agile management for evolving systems," in *Hot topics in autonomic computing (HotAC)*, Dublin, Ireland, May 2006. 30, 121, 124

Google Inc., "Google Apps service level agreement," 2010. [Online]. Available: http://www.google.com/apps/intl/en/terms/sla.html 66, 109

A. Gorbenko, V. Kharchenko, P. Popov, and A. Romanovsky, "Dependable composite web services with components upgraded online," in *Architecting Dependable Systems III*, R. de Lemos, C. Gacek, and A. Romanovsky, Eds. Springer-Verlag, LNCS 3549, 2005, pp. 92–121. 28

J. C. Gower, "A general coefficient of similarity and some of its properties," *Biometrics*, vol. 27, no. 4, pp. 857–871, Dec 1971. 38

J. Gray, "A census of Tandem system availability between 1985 and 1990," *IEEE Transactions on Reliability*, vol. 39, no. 4, pp. 409–418, Oct 1990. 2

A. Gupta and I. S. Mumick, "Maintenance of materialized views: Problems, techniques, and applications," *IEEE Data Engineering Bulletin*, vol. 18, no. 2, pp. 3–18, 1995. 56, 62

D. Gupta, "On-line software version change," Ph.D. dissertation, Indian Institute of Technology, Kanpur, 1994. 20

S. Hansell, "Glitch makes teller machines take twice what they give," *The New York Times*, 18 Feb 1994. 128, 130, 133, 142

J. Hart and J. D'Amelia, "An analysis of RPM validation drift," in *USENIX Large Installation System Administration Conference*, Philadelphia, PA, Nov 2002, pp. 155–166. 22

C. M. Hayden, E. A. Hardisty, M. Hicks, and J. S. Foster, "Efficient systematic testing for dynamically updatable software," in *ACM Workshop on Hot Topics in Software Upgrades*, Orlando, FL, Oct 2009. 24, 92, 150

A. Hume (AT&T Labs), Personal communication, 2010. 133

IBM Corporation, "An architectural blueprint for autonomic computing," White Paper, Jun 2006, 4th edition. [Online]. Available: http://www-01.ibm.com/software/tivoli/autonomic/pdfs/AC_Blueprint_White_Paper_4th.pdf 30

IBM Corporation, *WebSphere Extended Deployment Version 5.1 Information Center*, 2004. [Online]. Available: http://publib.boulder.ibm.com/infocenter/wxddoc51/index.jsp 118, 121

IBM Corporation, "Tivoli intelligent orchestrator," http://www-947.ibm.com/support/entry/portal/Overview/Software/Tivoli/Tivoli_Intelligent_Orchestrator, 2007. 112, 115, 118, 124

F. Kaashoek, B. Liskov, D. Andersen, M. Dahlin, C. Ellis, S. Gribble, A. Joseph, H. Levy, A. Myers, J. Mogul, I. Stoica, and A. Vahdat, "Report of the NSF workshop on research challenges in distributed computer systems," Dec 2005. 1

K. Kanoun and L. Spainhower, Eds., *Dependability Benchmarking for Computer System*. John Wiley & Sons, 2008. 91

L. Kaufman and P. J. Rousseeuw, *Finding Groups in Data: an Introduction to Cluster Analysis*, ser. Wiley Series in Probability and Mathematical Statistics. John Wiley & Sons, 1990. 38, 43

A. Keller, J. Hellerstein, L. Wolf, K. Wu, and V. Krishnan, "The CHAMPS system: Change management with planning and scheduling," in *Network Operations and Management Symposium*, Seoul, Korea, Apr 2004, pp. 395–408. 30, 112, 115, 124

L. Keller, P. Upadhyaya, and G. Candea, "ConfErr: A tool for assessing resilience to human configuration errors," in *International Conference on Dependable Systems and Networks*, Anchorage, AK, Jun 2008. 18, 19, 34, 37, 39, 47, 48, 104

J. Kephart and D. Chess, "The vision of autonomic computing," *IEEE Computer*, vol. 36, no. 1, pp. 41–50, Jan 2003. 30, 112

S. Kim, T. Zimmermann, J. E. James Whitehead, and A. Zeller, "Predicting faults from cached history," in *International Conference on Software Engineering*, Minneapolis, MN, May 2007, pp. 489–498. 36

C. Koch, "AT&T Wireless self-destructs," *CIO Magazine*, Apr 2004. [Online]. Available: http://www.cio.com/archive/041504/wireless.html 3

J. Kramer and J. Magee, "The evolving philosophers problem: Dynamic change management," *IEEE Transactions on Software Engineering*, vol. 16, no. 11, pp. 1293–1306, 1990. 8, 19, 26, 69

J. Kramer and J. Magee, "Dynamic configuration for distributed systems," *IEEE Transactions on Software Engineering*, vol. 11, no. 4, pp. 424–436, 1985. 26, 129

B. Krebs, "Cyber incident blamed for nuclear power plant shutdown," *The Washington Post*, 5 Jun 2008. [Online]. Available: http://www.washingtonpost.com/wp-dyn/content/article/2008/06/05/AR2008060501958.html 4

S. Kumar, V. Talwar, V. Kumar, P. Ranganathan, and K. Schwan, "vManage: Loosely coupled platform and virtualization management in data centers," in *International Conference on Autonomic computing*, Barcelona, Spain, Jun 2009, pp. 127–136. 113

N. LaBelle and E. Wallingford, "Inter-package dependency networks in open-source software," in *International Conference on Complex Systems*, Boston, MA, Jun 2006. 21

J. R. Levine, *Linkers and Loaders*. Morgan Kaufmann Publishers, 2000. 80

J. Leyden, "Windows Update keeps tabs on all system software," *The Register*, 28 Feb 2003. [Online]. Available: http://www.theregister.co.uk/2003/02/28/windows_update_keeps_tabs 5, 21, 156

Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai, "Have things changed now?: an empirical study of bug characteristics in modern open source software," in *ASPLOS Workshop on Architectural and System Support for Improving Software Dependability*, 2006, pp. 25–33. 32, 36

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, pp. 141–154. 146

B. Liskov, "Software upgrades in distributed systems," Keynote address at the ACM Symposium on Operating Systems Principles, Oct 2001. 10, 110

B. Liskov, "Distributed programming in Argus," *Communications of the ACM*, vol. 31, no. 3, pp. 300–312, 1988. 25

D. Lowell, Y. Saito, and E. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," in *International Conference on Architectural Support for Programming Languages and Operating Systems*, Boston, MA, Oct 2004, pp. 211–223. 2, 25, 50

S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou, "BugBench: A benchmark for evaluating bug detection tools," in *PLDI Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, IL, Jun 2005. 34

K. Magoutis, M. Devarakonda, N. Joukov, and N. G. Vogl, "Galapagos: Model-driven discovery of end-to-end application-storage relationships in distributed systems," *IBM Journal of Research & Development*, vol. 52, no. 4/5, p. 367, 2008. 5, 23, 93

K. Makris and R. A. Bazzi, "Immediate Multi-Threaded Dynamic Software Updates Using Stack Reconstruction," in *Proceedings of the USENIX '09 Annual Technical Conference*, June 2009. 24

B. Malik and D. Scott, "Best practices for continuous application availability," in *Gartner Data Center Conference*, Las Vegas, NV, Dec 2008. 2

F. Mancinelli, J. Boender, R. Di Cosmo, J. Vouillon, B. Durak, X. Leroy, and R. Treinen, "Managing the complexity of large free and open source package-based software distributions," in *International Conference on Automated Software Engineering*, Tokyo, Japan, Sep 2006, pp. 199–208. 22, 157

N. McNab and A. Bryan, "An implementation of the Linux software repository model for other operating systems," in *ACM Workshop on Hot Topics in Software Upgrades*, Orlando, FL, Oct 2009. 21, 23

D. Menascé, "TPC-W: A benchmark for e-commerce," *IEEE Internet Computing*, vol. 6, no. 3, pp. 83–87, May/Jun 2002. 71, 94, 99

Microsoft Corporation, "Perform a rolling upgrade from Windows 2000," TechNet Library, Jan 2005. [Online]. Available: http://technet.microsoft.com/en-us/library/cc738005(WS.10).aspx 8, 29, 48, 86, 127

Microsoft Corporation, "Microsoft SQL Server 2008 R2," http://www.microsoft.com/sqlserver/en/us/default.aspx, 2010. 61

Microsoft Developer Network, *Windows Update Agent*, 2001, http://msdn2.microsoft.com/en-us/library/aa387099.aspx. 21, 146

L. E. Moser, P. M. Melliar-Smith, and P. Narasimhan, "Consistent object replication in the Eternal system," *Theory and Practice of Object Systems*, vol. 4, no. 2, pp. 81–92, 1998. 25

L. Moser, P. Melliar-Smith, P. Narasimhan, L. Tewksbury, and V. Kalogeraki, "Eternal: fault tolerance and live upgrades for distributed object systems," in *Information Survivability Conference and Exposition*, Hilton Head, SC, Jan 2000, pp. 184 – 196. 8

N. Nagappan, A. Zellery, T. Zimmermannz, K. Herzigx, and B. Murphy, "Change bursts as defect predictors," in *International Symposium on Software Reliability Engineering*, San Jose, CA, Nov 2010. 32

K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen, "Understanding and dealing with operator mistakes in Internet services," in *USENIX Symposium on Operating Systems Design and Implementation*, San Francisco, CA, Dec 2004, pp. 61–76. 18, 27, 29, 34, 35, 36, 39, 93, 94, 102

P. Narasimhan, "Transparent fault-tolerance for CORBA," Ph.D. dissertation, University of California, Santa Barbara, 1999. 17, 25, 62, 86

I. Neamtiu and M. Hicks, "Safe and timely dynamic updates for multi-threaded programs," in *ACM Conference on Programming Language Design and Implementation*, Dublin, Ireland, Jun 2009. 92

I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol, "Practical dynamic software updating for C," in *ACM Conference on Programming Language Design and Implementation*, Ottawa, Canada, Jun 2006, pp. 72–83. 7, 19, 23, 24, 92

P. Neumann *et al.*, "America Offline," *The Risks Digest*, vol. 18, no. 30–31, 8–9 Aug 1996, http://catless.ncl.ac.uk/Risks/18.30.html. 4

L. Northrop *et al.*, *Ultra-Large-Scale Systems: The Software Challenge of the Future*. SEI Carnegie Mellon University, Jun 2006. 32

Oxford English Dictionary, 2nd ed. Oxford University Press, 1989, http://www.oed.com. 3, 67

Office of Government Commerce, "Information technology infrastructure library (ITIL)," 2001. 48

Office of Government Commerce, *Service Transition*, ser. Information Technology Infrastructure Library (ITIL), 2007. 8, 28, 47, 48, 88, 92

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. 18, 27, 28, 29, 34, 35, 36, 39, 48, 85, 93, 94, 102, 130, 151

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. 2, 18, 29, 34, 37, 39, 130

Oracle Corporation, "Database rolling upgrade using Data Guard SQL Apply," Maximum Availability Architecture White Paper, Dec 2008. [Online]. Available: http://www.oracle.com/technology/deploy/availability/pdf/maa_wp_10gr2_rollingupgradebestpractices.pdf 8, 29, 48, 50, 75, 86, 88, 127, 131

Oracle Corporation, "Oracle Database 11g Release 2," http://www.oracle.com/technetwork/database/enterprise-edition/overview/index.html, 2009. 61, 62

Oracle Corporation, *Oracle Real Application Cluster 10g*, 2005. 118, 121

M. P. Papazoglou and D. Georgakopoulos, "Service-oriented computing," *Communications of the ACM*, vol. 46, no. 10, pp. 24–28, 2003. 109

P. Patel, A. Whitaker, D. Wetherall, J. Lepreau, and T. Stack, "Upgrading transport protocols using untrusted mobile code," in *Symposium on Operating Systems Principles*, Bolton Landing, NY, Oct 2003, pp. 1–14. 153

D. Patterson, "A simple way to estimate the cost of downtime," in *USENIX Large Installation System Administration Conference*, Philadelphia, PA, Nov 2002, pp. 185–188. 50, 66

C. Peltz, "Web services orchestration and choreography," *IEEE Computer*, vol. 36, pp. 46–52, 2003. 109

S. Pertet and P. Narasimhan, "Proactive recovery in distributed CORBA applications," in *International Conference on Dependable Systems and Networks*, Florence, Italy, Jun 2004. 111

D. Petrou (Google), Personal communication, 2010. 153

M. Pinedo, *Scheduling: Theory, Algorithms and Systems*, 2nd ed. Prentice Hall, 2002. 119

S. Potter and J. Nieh, "Reducing downtime due to system maintenance and upgrades," in *USENIX Large Installation System Administration Conference*, San Diego, CA, Dec 2005, pp. 47–62. 25

Quest Software, Inc., "Shareplex for Oracle," http://www.quest.com/Quest_Site_Assets/PDF/SPO_752_ReleaseNotes.htm, 2010. 62

J. Reason, *Human Error*. Cambridge University Press, 1990. 37, 39

D. Reiss, "Release management and software deployment at Facebook," Keynote adress at the ACM Workshop on Hot Topics in Software Upgrades, Oct 2009. 63, 66, 128, 130, 132, 133, 142, 146, 152

J. Rellermeyer, G. Alonso, and T. Roscoe, "R-OSGi: Distributed applications through software modularization," in *ACM/IEEE/IFIP Middleware Conference*, Newport Beach, CA, Dec 2007. 26

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. 8, 26

J. Rexford, Personal communication, 2007. 50

D. Roșu and A. Dan, "Managing end-to-end lifecycle of global service policies," in *International Conference on Service-Oriented Computing*. Amsterdam, The Netherlands: Springer-Verlag, LNCS 3826, Dec 2005, pp. 570–575. 119

O. Rütti, P. T. Wojciechowski, and A. Schiper, "Structural and algorithmic issues of dynamic protocol update," in *International Symposium on Parallel and Distributed Processing*, Rhodes Island, Greece, Apr 2006. 153

K. Salem, K. Beyer, B. Lindsay, and R. Cochrane, "How to roll a join: Asynchronous incremental view maintenance," *SIGMOD Record*, vol. 29, no. 2, pp. 129–140, Jun 2000. 74

F. Schmuck and R. Haskin, "GPFS: A shared-disk file system for large computing clusters," in *USENIX Conference on File and Storage Technologies*, Monterey, CA, Jan 2002, pp. 231–244. 60

F. Schmuck (IBM Research), Personal communication, 2010. 60

M. Segal, "Online software upgrading: new research directions and practical considerations," in *Computer Software and Applications Conference*, Oxford, England, Aug 2002, pp. 977–981. 8, 24, 92, 127, 151

M. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, vol. 10, no. 2, pp. 53–65, Mar 1993. 7, 20, 24, 26, 92

M. E. Segal and O. Frieder, "Dynamic program updating: A software maintenance technique for minimizing software downtime," *Journal of Software Maintenance: Research and Practice*, vol. 1, no. 1, pp. 59–79, 1989a. 24, 26

M. E. Segal and O. Frieder, "Dynamically updating distributed software: supporting change in uncertain and mistrustful environments," in *IEEE Conference on Software Maintenance*, Oct 1989b, pp. 254–261. 26, 129

L. Sha, R. Rajkumar, and M. Gagliardi, "Evolving dependable real-time systems," in *IEEE Aerospace Applications Conference*, Aspen, CO, 3–10 1996, pp. 335–46. 28

G. N. Silva, *APT HOWTO*, Aug 2005. [Online]. Available: http://www.debian.org/doc/manuals/apt-howto/index.en.html 5, 7, 21, 22, 156

J. Sliwerski, 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. 146

I. Sommerville, *Software Engineering*, 8th ed. Addison-Wesley, 2007. 135

I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan, "Chord: A scalable peer-to-peer lookup service for internet applications," in *ACM Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications*, San Diego, CA, Aug 2001, pp. 149–160. 86

G. Stoyle, M. Hicks, G. Bierman, P. Sewell, and I. Neamtiu, "*Mutatis Mutandis*: Safe and predictable dynamic software updating," *ACM Transactions on Programmin Languages and Systems*, vol. 29, no. 4, 2007. 20

M. Sullivan and R. Chillarege, "Software defects and their impact on system availability-a study of field failures in operating systems," in *Fault-Tolerant Computing Symposium*, Montreal, Canada, Jun 1991, pp. 2–9. 32, 34

Y. Sun and A. Couch, "Global impact analysis of dynamic library dependencies," in *USENIX Large Installation System Administration Conference*, San Diego, California, Dec 2001, pp. 145–150. 5

E. B. Swanson, "The dimensions of maintenance," in *International Conference on Software Engineering*, San Francisco, CA, 1976, pp. 492–497. 129

A. Tai, K. Tso, L. Alkalai, S. Chau, and W. Sanders, "Low-cost error containment and recovery for onboard guarded software upgrading and beyond," *IEEE Transactions on Computers*, vol. 51, no. 2, pp. 121–137, Feb 2002. 28

Y.-L. Tan, T. Wong, J. D. Strunk, and G. R. Ganger, "Comparison-based file server verification," in *Proceedings of the annual conference on USENIX Annual Technical Conference*, Anaheim, CA, 2005, pp. 121–133. 27, 28

E. Tempero, G. Bierman, J. Noble, and M. Parkinson, "From Java to UpgradeJ: an empirical study," in *ACM Workshop on Hot Topics in Software Upgrades*, Nashville, TN, Oct 2008. 24

L. Tewksbury, L. Moser, and M. Melliar-Smith, "Live upgrades of CORBA applications using object replication," in *International Conference on Software Maintenance*, Florence, Italy, Nov 2001, pp. 488–497. 25, 26, 129

E. Thereska, M. Abd-El-Malek, J. J. Wylie, D. Narayanan, and G. R. Ganger, "Informed data distribution selection in a self-predicting storage system," in *International Conference on Autonomic Computing*, Dublin, Ireland, Jun 2006. 30, 111, 118

L. C. Toy, "Large-scale real-time program retrofit methodology in AT&T 5ESS switch," in *Reliable computer systems: design and evaluation*, 2nd ed., D. P. Siewiorek and R. S. Swarz, Eds. Digital Press, 1992, pp. 574–586. 2, 23

B. Treynor, "More on today's Gmail issue," The Official Gmail Blog, Sep 2009. [Online]. Available: http://gmailblog.blogspot.com/2009/09/more-on-todays-gmail-issue.html 4

J. Tucek, W. Xiong, and Y. Zhou, "Efficient online validation with delta execution," in *International Conference on Architectural Support for Programming Languages and Operating Systems*. New York, NY, USA: ACM, 2009, pp. 193–204. 27

C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner, "OPIUM: Optimal package install/uninstall manager," in *International Conference on Software Engineering*, Minneapolis, MN, May 2007, pp. 178–188. 7, 22, 48

G. Urdaneta, G. Pierre, and M. van Steen, "Wikipedia workload analysis for decentralized hosting," *Computer Networks*, vol. 53, no. 11, pp. 1830–1845, July 2009. 53

S. Vermeulen, G. Goodyear, R. Marples, D. Robbins, C. Houser, and J. Alexandratos, *Gentoo Handbook*, May 2007. [Online]. Available: http://www.gentoo.org/doc/en/handbook/handbook-x86.xml 5, 21, 156

S. Vinoski, "Convenience over correctness," *IEEE Internet Computing*, vol. 12, no. 4, pp. 89–92, 2008. 134

J. Waldo, "The end of protocols," *Sun Microsystems Java Developer Connection*, June 2000. [Online]. Available: http://java.sun.com/developer/technicalArticles/jini/protocols.html 153

J. Waldo (VMware), Personal communication, 2010. 153

Y.-M. Wang, C. Verbowski, J. Dunagan, Y. Chen, Y. Chun, H. J. Wang, and Z. Zhang, "STRIDER: A black-box, state-based approach to change and configuration management and support," in *USENIX Large Installation System Administration Conference*, San Diego, CA, Oct 2003, pp. 159–172. 7

R. L. Wears, R. I. Cook, and S. J. Perry, "Automation, interaction, complexity, and failure : A case study," *Reliability Engineering and System Safety*, vol. 91, no. 12, pp. 1494–1501, Dec 2006. 4

Wikimedia Foundation, "MediaWiki 1.5 upgrade," 2005. [Online]. Available: http://meta.wikimedia.org/wiki/MediaWiki_1.5_upgrade 57

A. Williams (A.T. Kearney), Personal communication, 2009. 51, 60

L. Wong, N. S. Arora, L. Gao, T. Hoang, and J. Wu, "Oracle Streams: A high performance implementation for near real time asynchronous replication," in *ICDE*, Shanghai, China, Apr 2009, pp. 1363–1374. 62

S. Yau and J. Collofello, "Some stability measures for software maintenance," *IEEE Transactions on Software Engineering*, vol. 6, no. 6, pp. 545–552, Nov 1980. 32

Q. Zhang, A. Riska, W. Sun, E. Smirni, and G. Ciardo, "Workload-aware load balancing for clustered Web servers," *International Symposium on Parallel and Distributed Processing*, vol. 16, no. 3, pp. 219–233, Mar 2005. 111

W. Zheng, R. Bianchini, and T. D. Nguyen, "Automatic configuration of Internet services," in *European Conference in Computer Systems (EuroSys)*, Lisbon, Portugal, Mar 2007, pp. 219–229. 48, 94

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. 29, 130

I. Zolti (Accenture), Personal communication, 2006. 66, 87

Index