Dependable, Online Upgrades in Enterprise Systems

Tudor Dumitraş
Carnegie Mellon University
tudor@cmu.edu

Abstract

Software upgrades are unreliable, often causing downtime or data loss. I propose Imago, an approach for removing the leading causes of upgrade failures (broken dependencies) and of planned downtime (data migrations). While imposing a higher resource overhead than previous techniques, Imago is more dependable and easier to use correctly.

Categories and Subject Descriptors K.6.3 [*Management of Computing and Information Systems*]: Software Management

General Terms Management, Reliability

Keywords software upgrades, online upgrades, dependability, hidden dependencies, data migration

1. Introduction

In the presence of new user requirements, modified deployment environments and bug fixes, enterprise software-systems must evolve continuously. Aside from the known challenges of software evolution [4], a large body of anecdotal evidence suggests that the upgrade procedures used, in practice, for accomplishing the evolution are failure-prone and often cause downtime, data corruption or latent errors. I present Imago,[1] a mechanism for *improving the availability of a system-under-upgrade in the fault-free case* – by performing an online upgrade – *and in the faulty case* – by isolating the new version from the upgrade operations and by performing an atomic upgrade, end-to-end.

While current 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 concluded that, on average, 8.6% of upgrades fail, with some system administrators reporting failure rates up to 50% [2]. The survey identified broken dependencies and altered system-behavior as the leading causes of upgrade failure, followed by bugs in the new version. This suggests that *most upgrade failures are not due to software defects, but to faults that affect the upgrade procedure*. Furthermore, 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 migrations currently impose downtime on upgrade*. Such planned outages typically last from tens of hours to several days, *i.e.*, twice as long as unplanned ones.

[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.

OOPSLA 2009, October 25–29, 2009, Orlando, Florida, USA.
ACM 978-1-60558-768-4/09/10.

Previous research has concentrated on performing upgrades *in-place* [2, 6], which requires tracking the complex dependencies among the distributed components of the version that is already deployed, and on supporting *mixed versions* [1, 5, 6], which interact and synchronize their states in the presence of a live workload. Ensuring the correctness of mixed-version interactions requires manual assistance from the programmer [1, 6] (*e.g.*, establishing constraints to prevent old code from accessing new data [6]). Because some dependencies cannot be inferred automatically, upgrading approaches based on dependency-tracking ultimately rely on a time-intensive and error-prone manual process (*e.g.*, an in-depth pointer analysis [5, 6]), to understand the nature and depth of the dependency chain. Moreover, the problem of resolving the dependencies of a component before deployment is NP-complete. This suggests that, because of the increasing complexity of enterprise systems, upgrading approaches that rely on tracking dependencies could become computationally infeasible in the future.

Because of these fundamental limitations, industry best-practices recommend “rolling upgrades,” which upgrade-and-reboot one node at a time, in a wave rolling through the distributed system. This in-place, mixed-version approach is believed to reduce the risks of upgrading because localized failures might not affect the entire distributed system [7, 8]. However, recent commercial products for rolling upgrades do not guarantee that the interactions among mixed versions are safe and leave these concerns to the application developers [8]. Through two empirical studies, I show that rolling upgrades (i) are failure-prone – because the upgrade is not an atomic operation and it risks breaking *hidden dependencies* in the system-under-upgrade – and (ii) do not support the data migrations required for upgrading a popular Internet service.

2. Goal statement

I propose to improve the dependability of enterprise upgrades by removing the leading cause of upgrade failures – broken dependencies – and by providing a solution for the leading cause of planned downtime – data migrations. A dependable, online upgrade should have three properties:

- **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).
- **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.
- **Fidelity**: the upgrade testing must reproduce realistically the conditions of the deployment environment.

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 and fidelity properties imply that the system must not include mixed, interacting versions, which synchronize their states in the back-end and exhibit emerging behaviors that are difficult to validate through offline testing. These properties enable long-running data migrations 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. Moreover, because it does not require correctness constraints for the mixed-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 requires additional hardware and storage resources. The high-level goals of my research are to study these trade-offs and to evaluate how close can we get to the zero-downtime ideal.

Non-goals. My research does not aim to provide support for minor upgrades, such as fine-grained bug fixes or security patches, to perform upgrades in-place, without the need for additional resources, or to eliminate upgrade failures that are due to software defects.

3. Technical approach

Imago is a prototype that implements the isolation, atomicity and fidelity properties for enterprise-system upgrades.

Feasibility study. To understand why upgrades fail, I analyze data from three independent sources [3]: a user study of system administration tasks in an e-commerce system, a survey of database administrators and a field study of bug reports for the Apache web server. This analysis includes procedural and configuration errors that occur during upgrades and excludes software defects. 85% of the faults analyzed break a hidden dependency (*e.g.*, shared-library conflict, wrong configuration, database-schema mismatch). Using statistical cluster-analysis, I establish an *upgrade-centric fault model* with four categories: (1) simple configuration or procedural 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) errors that render the persistent-data partially unavailable. These upgrade faults cause outages or degrade the performance and cannot be easily masked using existing techniques.

I also study the upgrade history of Wikipedia, one of the ten most popular websites to date, in order to determine the *common reasons for planned downtime*. Incompatible changes to the database schema (*e.g.* dropping/renaming tables) prevent rolling upgrades and require upgrading the schema and the business logic in an atomic step, in order to avoid Type 4 upgrade faults. Long-running data conversions compete with the live workload and might overload the database. Data dependencies (*e.g.*, resulting from table joins) are hard to synchronize in response to updates issued by the live workload. Because of these reasons, 50 out of the 55 possible upgrades of Wikipedia's business logic would require planned downtime.

Dependable, online upgrades with Imago. Imago achieves the isolation property by installing the new version in a *parallel universe* – a logically distinct collection of resources, realized either using additional hardware or through virtualization. While the old version continues to service the live workload, Imago opportunistically transfers the persistent data into the new version. Imago reads the old version's data-store without locking objects and regulates its data-transfer rate in order to avoid interfering with the client requests. Imago intercepts the live-request flow at two points in the old version: the *ingress* point, which receives requests from the live version (*e.g.* the front-end proxy), and the *egress* point, which stores the persistent data (*e.g.* the master database). The egress interceptor monitors the data objects updated by the live workload and (re-)schedules them for transfer. The ingress interceptor implements the coordinated switchover to the new version. Imago supports a series of iterative testing phases after completing the data-transfer, and it provides the opportunity for testing the new version online, using the live requests recorded by the ingress interceptor, before exposing the upgrade to the clients. After adequate testing, Imago switches over to the new version, completing the upgrade as an atomic operation. Because it avoids overloading the production system, Imago also enables the integration of long-running data conversions in an online upgrade. Imago requires additional resources only for implementing and testing the online upgrade, which suggests that storage and compute cycles could be leased, for the duration of the upgrade, from existing cloud-computing infrastructures in order to provide upgrades-as-a-service.

Experimental evaluation. Fault-injection experiments suggest that rolling upgrades are vulnerable to upgrade faults because they create system states with mixed versions, where it is easy to break hidden dependencies. Contrary to the conventional wisdom, these faults can have a global impact on the system-under-upgrade, such as outages, throughput- or latency-degradations, security vulnerabilities or latent errors. For instance, the database represents a single point of failure for an in-place upgrade, and any Type 4 fault leads to an upgrade failure. In contrast, Imago eliminates the single-points-of-failure for Types 1–4 of upgrade faults by avoiding an in-place upgrade and by isolating the old version from the upgrade operations. Imago is only vulnerable to latent errors that are introduced when configuring the new version and that remain undetected during testing. Compared with a rolling upgrade, Imago reduces the expected unavailability due to upgrade faults (result significant at the $p = 0.01$ level). Moreover, Imago supports all the data migrations recorded in Wikipedia's upgrade history, without degrading the throughput or response time of the production system during the upgrade.

4. Conclusions

Using a novel, upgrade-centric fault model, I show that current approaches for upgrading enterprise systems are failure-prone because the upgrade is not an atomic operation and because it risks breaking hidden dependencies among the distributed system-components. I present Imago, which performs upgrades dependably despite hidden dependencies in the system-under-upgrade. Additionally, Imago supports online upgrades with complex data migrations and allows testing the new version in its operational environment, without overloading the production system. Because it avoids dependency-tracking and mixed-version interactions, Imago will likely be easier to use correctly than existing techniques.

References

[1] C. Boyapati *et al.* Lazy modular upgrades in persistent object stores. In *Object-Oriented Programing, Systems, Languages and Applications*, pages 403–417, Anaheim, CA, Oct 2003.

[2] O. Crameri *et al.* Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Symposium on Operating Systems Principles*, pages 221–236, Stevenson, WA, Oct 2007.

[3] T. Dumitraş and P. Narasimhan. Why do upgrades fail and what can we do about it? In *ACM/IEEE/IFIP Middleware Conference*, Urbana Champaign, IL, Nov-Dec 2009.

[4] T. Mens *et al.* Challenges in software evolution. In *Workshop on Principles of Software Evolution*, pages 13–22, Lisbon, Portugal, Sep 2005.

[5] L. Moser *et al.* Eternal: fault tolerance and live upgrades for distributed object systems. In *Information Survivability Conference and Exposition*, pages 184 – 196, Hilton Head, SC, Jan 2000.

[6] I. Neamtiu, M. Hicks, G. Stoyle, and M. Oriol. Practical dynamic software updating for C. In *ACM Conference on Programming Language Design and Implementation*, pages 72–83, Ottawa, Canada, Jun 2006.

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

[8] Oracle Corporation. Rolling upgrades of stateful J2EE applications in Oracle Application Server. White Paper, Aug 2005.