ACM SRC Entry: Tudor Dumitraş (http://www.ece.cmu.edu/~tdumitra)
Research Advisor: Priya Narasimhan
Category: Graduate, First Place in OOPSLA'09 SRC

Improving the End-to-End Dependability of Distributed Systems through AIR Software Upgrades

Tudor Dumitraş
Carnegie Mellon University

tdumitra@ece.cmu.edu

Managing change is one of the essential challenges for ensuring the dependability of software systems [Brooks, 1987]. 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 or latent errors.

My research makes four contributions:

- Through two empirical studies, I identify the *leading causes of upgrade failure*—breaking hidden dependencies—*and of planned downtime*—changing database schemas;
- I introduce 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;
- I describe the design of *Imago*,[1] a system that guarantees the AIR properties and that incorporates end-to-end mechanisms for performing major software upgrades in large-scale distributed systems;
- I propose an *analytical risk-assessment* methodology for estimating the impact of providing relaxed versions of the AIR properties.

The key idea is to isolate the production system from the upgrade operations in order to avoid breaking hidden dependencies. The end-to-end upgrade is an atomic operation, executed online even when performing complex schema and data conversions. Imago harnesses the opportunities provided by cloud computing technologies to simplify major enterprise-system upgrades and to improve their dependability. This approach separates the functional aspects of the upgrade (e.g., data migration) from the mechanisms for online upgrade (e.g., atomic switchover), enabling an *upgrades-as-a-service* model.

Goal Statement

I propose to *improve the dependability of distributed systems* by (i) removing the leading cause of upgrade failures and (ii) providing a solution for the leading cause of upgrade-related planned downtime. I focus on upgrading enterprise distributed systems end-to-end, which requires the coordinated replacement of multiple system components and the conversion of the persistent data to the new format.

Dependability improvements usually come at a cost. In my research, I make a *fundamental trade-off*: in order to reduce the planned and unplanned downtime, my approach imposes a higher resource overhead than previous techniques. 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 [Zolti, 2006; Downing, 2008; Reiss, 2009; Choi, 2009]. The *high-level goals* of my research are to study such trade-offs, to assess the effort required to implement and coordinate a complex online-upgrade 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 upgrade distributed systems in a fully-transparent manner.

Problem Identification

Software upgrades are unavoidable in distributed enterprise systems. For example, business reasons sometimes mandate switching vendors; responding to customer expectations and conforming with government regulations can require new functionality. Many enterprises can no longer afford to incur the high cost of downtime and must perform such upgrades online, without stopping their systems. However, recent studies and a large body of anecdotal evidence suggest that distributed-system upgrades remain unreliable and often induce unplanned or planned downtime.

For example, in 2003, the upgrade of a customer relationship management (CRM) system at AT&T Wireless created a ripple effect that disabled several key systems, affecting 50,000 customers per week [Koch, 2004]. The complexity of dependencies on 15 legacy back-

end systems was unmanageable, the integration could not be tested in advance in a realistic environment, and rollback became impossible because enough of the old version had not been preserved. 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. 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 [Lowell et al., 2004; Downing, 2008].

Causes of unplanned downtime

I establish an *upgrade-centric fault model*, by analyzing data from three independent sources [Dumitraş 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 focuses on unavoidable human errors in the upgrade procedure, which break *hidden dependencies* (e.g., specifying wrong service locations, creating database-schema mismatches, introducing shared-library conflicts) in the system under upgrade.

Through statistical clustering techniques (widely used in the natural sciences for creating taxonomies of living organisms) I show that there are four common types of upgrade faults (Figure 1):

1. *Simple configuration or procedural errors*, e.g. typos in a configuration file;
2. *Semantic configuration errors*, which indicate a misunderstanding of the configuration directives used;
3. *Broken environmental dependencies*, e.g. library or port conflicts in the deployment environment;
4. *Data-access errors*, which render the persistent data partially unavailable.

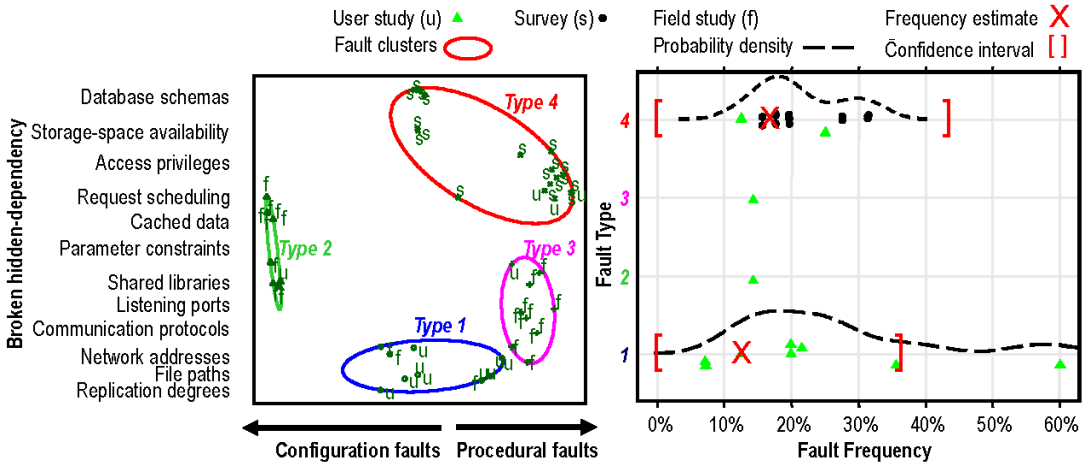

Figure 1. Upgrade-centric fault model. Statistical cluster analysis (left) identifies four common types of upgrade faults. The survey and the user study also provide information about the distribution of fault-occurrence rates (right).

These faults represent the leading causes of upgrade failure in distributed systems and cannot be easily masked using existing techniques. I also estimate that faults of Types 1 and 4 occur in 14.6% and 18.7% of upgrades, respectively.

Causes of planned downtime

Even successful upgrades often require planned downtime. 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 [Dumitraş and Narasimhan, 2009b].

Out of the 55 possible upgrades of Wikipedia's business logic, 50 would impose planned downtime (see Figure 2). There are four common causes for such downtime:

- *Incompatible database-schema changes* (e.g. renaming or partitioning tables) require upgrading the schema and the business logic in an atomic step, in order to avoid Type 4 upgrade faults.
- *Long-running data conversions* (e.g., converting the entire encyclopedia to the UTF-8 character set) compete with the live workload and might overload the database.
- *Data dependencies* (e.g., resulting from table joins) are hard to synchronize on-the-fly, in response to updates issued by the live workload.

- *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.

For Wikipedia, incompatible schema changes and computationally-intensive data conversions represent the leading causes of planned downtime. For example, Wikipedia's most complex upgrade, performed in 2004, required migrating the content and meta-data of all current and past article revisions into three new database tables. During this upgrade, Wikipedia was locked for editing, and the schema was converted to the new version in approximately 22 hours. Conversely, complex database changes are often avoided because they might impose downtime, even when this amounts to rejecting user-requested features.

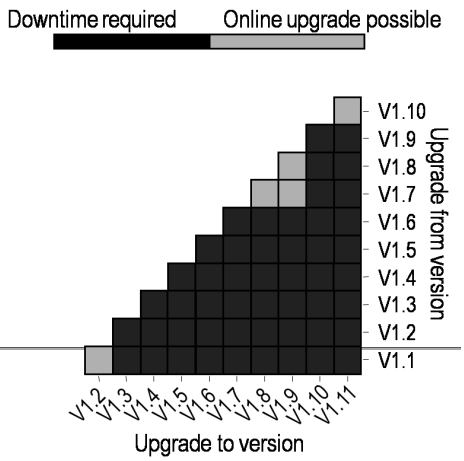

Figure 2. Planned downtime in Wikipedia.

Background and Related Work

Industry trends suggest that online upgrades are currently needed in large-scale distributed systems, such as electrical utilities, assembly-line manufacturing, customer support, e-commerce and online banking [Choi, 2009]. Previous research has focused on upgrading individual components of distributed systems rather than performing end-to-end upgrades. For example, dynamic software updating [Segal and Frieder,1993; Neamtiu et al., 2006] provides mechanisms for modifying a program on-the-fly and performs a single-node online upgrade where the entire system state is loaded in memory. In contrast, research on database-schema mapping and evolution [Ferrandina et al., 1995; Curino et al., 2008] concentrates on migrating persistent data stored in a database and provides mechanisms commonly used for planning offline upgrades. The approaches proposed for upgrading distributed systems [e.g., Bloom, 1983; Kramer and Magee, 1985; Ajmani et al., 2006] 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-life distributed systems are not based on a single, homogeneous framework. While these systems provide hardware and software redundancy, and are engineered to tolerate outages of individual components, they have more complex dependencies among the heterogeneous system components. Industry best practices [for example ITIL, 2007] recommend deploying the new version gradually, through "rolling upgrades" that upgrade-and-reboot each node at a time, in a wave rolling through the distributed system, and that place the system in a state with mixed versions. Current commercial products for rolling upgrades provide no way of determining if the interactions among mixed versions are safe and leave these concerns to the application developers [Oracle, 2008].

With these approaches, downtime is unavoidable. My empirical studies described above suggest that current upgrade mechanisms often induce unplanned downtime, by breaking hidden dependencies in the system under upgrade, and that they cannot always prevent planned downtime, in the the presence of complex schema changes.

Technical Approach

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 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 production system (including its performance, configuration settings and ability to access the data objects).
- **Runtime-testing**: The upgraded system must be tested under operational conditions.

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.

In this manner, the AIR properties *eliminate the leading causes of both planned and unplanned downtime* due to software upgrades.

Dependable, online upgrades with Imago

I demonstrate the feasibility of the AIR properties through the design and implementation of a system called Imago [Dumitraş and Narasimhan, 2009a]. Imago achieves isolation by installing the new version in a *parallel universe*—a logically distinct collection of resources, realized either using additional hardware or through virtualization (see Figure 3). While the old version continues to service the live workload, Imago opportunistically transfers the persistent data into the new version and converts it into the appropriate format. 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.

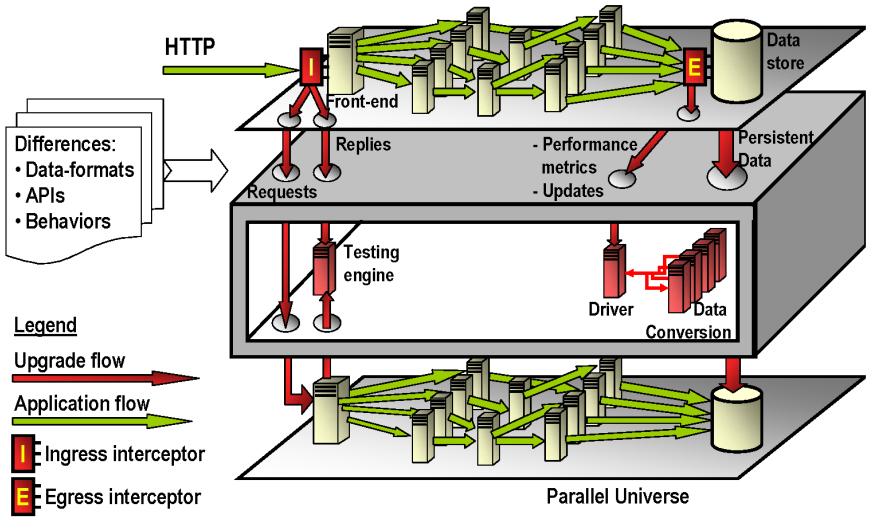

Figure 3. Architecture of Imago.

Imago intercepts the live-request flow at a few key points in the production system: 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.

Imago's main disadvantage is the resource overhead imposed by the parallel universe. I address this problem in two ways: (i) by leasing resources on-demand, from cloud-computing infrastructures, in order to provide upgrades-as-a-service and (ii) by modeling analytically the dependability of upgrades with relaxed AIR properties.

Upgrades-as-a-service

Imago requires additional resources only during the upgrade. This suggests that storage and compute cycles could be leased, for the duration of the upgrade, from existing cloud-computing infrastructures (e.g. the Amazon Web Services).

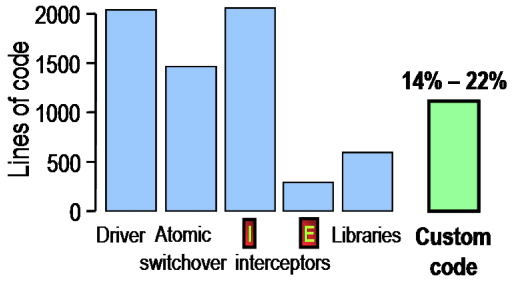

Figure 4. Imago's code base.

Current approaches for online upgrade must transfer the persistent state, while preserving the dependencies among the distributed components of the system under upgrade. Moreover, when gradually replacing an existing infrastructure (e.g., through a rolling upgrade), the potential interactions among mixed version must be carefully accounted for.

Imago must transfer the persistent state as well, but it is unaffected by broken dependencies or about mixed-version interactions. This allows designing generic upgrade mechanisms, which can be reused for upgrading different systems. The application-specific routines, which correspond the data conversions, constitute only 14%–22% of Imago's code (see Figure 4), and they would also be needed when performing an offline upgrade.

This enables an *upgrades-as-a-service* model [Dumitraş and Narasimhan, 2010]. Imago introduces a separation of concerns between the functional aspects of the upgrade (e.g. state transfer)—which correspond to the custom code that must be written for each upgrade—and the mechanisms required for performing an online upgrade (e.g., live-workload interception, atomic switchover)—which correspond to the reusable components.

Upgrade dependability with relaxed AIR properties

For some systems it might not be feasible to strictly enforce the AIR properties. In these situations, we must reason about the impact of relaxing these properties on system dependability.

For example, in systems that span multiple administrative domains and communicate via asynchronous messaging, the whole system cannot be upgraded on a uniform schedule. Relaxing the upgrade-atomicity property creates mixed-version states, where the system is vulnerable to a new type of race condition. A *mixed-version race* occurs during rolling upgrades when the first tier, running the new version, sends a callback that is dispatched to a second-tier server still running the old version (see Figure 5). Because the callback was intended for the new version, the server might throw an exception or might cause a silent corruption. This can occur, for example, in Web

2.0 applications using AJAX callbacks [Reiss, 2009]—where the first tier is the client's browser, loading the new version of the Javascript application code upon the initial request, and the second tier is web front-end of the site—or in systems using cloud-computing resources. In 1994, a similar upgrade in the data center of a bank caused each ATM withdrawal to be deducted twice from the customer's account, adding up to a $15M loss [Hansell, 1994].

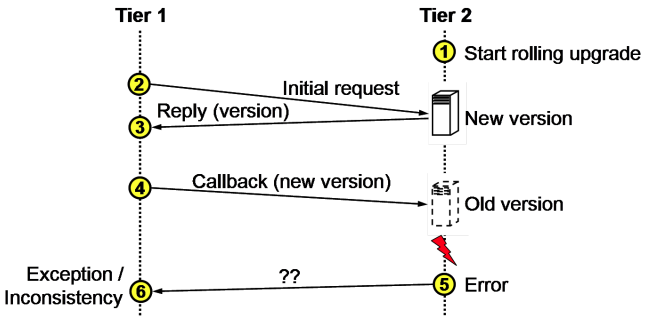

Figure 5. Mixed-version race.

By understanding the sequence of events that leads to mixed-version races, I propose an analytical model [Dumitraş et al., 2010] that estimates and compares the expected impact of two decisions: *to upgrade* and *not to upgrade*. The impact estimation is based on a uniform labeling system, which covers the severity of known bugs in the old version, the criticality of new feature requests, and the severity of inconsistencies due to mixed-version races. The probability of exposing an inconsistency varies during a rolling upgrade. If the application makes *c* callbacks per request and *i* out of *N* servers have been upgraded, the conditional probability of causing an inconsistency, given that a callback has new semantics, is:

$$P_{inc}(i) = \frac{i}{N} \cdot \left(1 - \left(\frac{i}{N}\right)^{c}\right)$$

Using this equation, I compute the average and the maximum risk of exposing an inconsistency during an upgrade. These estimations are included in an online risk-assessment tool, available at `http://orchestrate.cs.vt.edu:8080/examples/servlets/update.html`.

Dependability Evaluation

Most of the previous upgrade mechanisms have been evaluated by focusing on the types of changes supported or the overhead imposed, rather than the upgrade dependability. My evaluation of Imago seeks to answer three questions:

- What runtime overhead does Imago impose during a successful upgrade?
- Does Imago reduce the planned downtime, in the absence of upgrade faults?
- Does Imago reduce the unplanned downtime, in the presence of upgrade faults of types 1–4?

Imago adds, on average, 5 µs per request to the system's latency, well below the 50 ms threshold for human perception. A breakdown of this overhead per component shows that most of the delay is due to ingress interceptor (see Figure 6). The duration of the upgrade depends on the live workload, because Imago adjusts its transfer rate to prevent overloading the production system. Even under a flash-crowd scenario, where the system is severely overloaded, the data transfer will complete eventually if it can operate for 45 min per day.

Imago supports all the database schema changes that have imposed downtime for Wikipedia. These data transformations do not prevent the live workload from accessing the system during the upgrade. However, the atomic switchover to the new version requires planned downtime.[2] This downtime is on the order of minutes (<4 min, for systems with Apache or JBoss in the middle tier) rather than tens of hours or days. Moreover, the time needed to switch over does not increase indefinitely with the incoming load because the front-end servers already perform admission control and allow only a limited number of concurrent requests in the system.

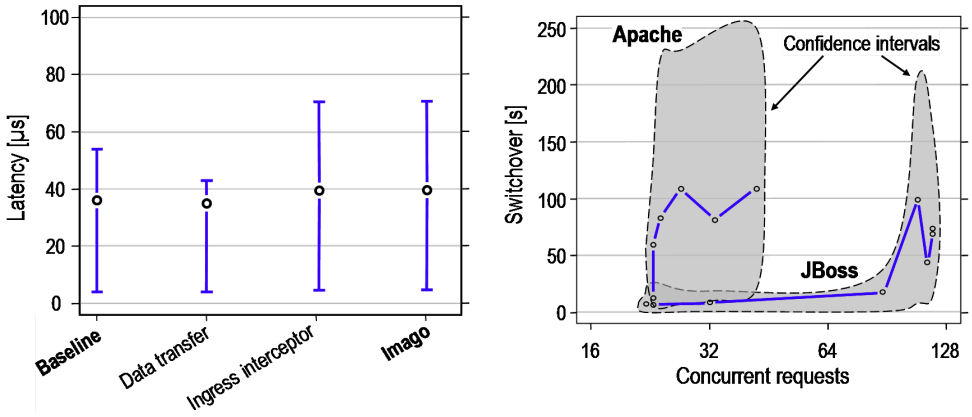

Figure 6. Runtime overhead (left) and planned downtime (right) imposed by Imago.

Fault-injection experiments, driven by my upgrade-centric fault model (Figure 7), 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 belief promoted by best-practice recommendations [e.g. ITIL, 2007], these localized faults can have a global impact on the system under upgrade, such as outages, throughput- or latency-degradations, security vulnerabilities or latent errors. For example, the database

represents a single point of failure for a rolling upgrade, and most Type 4 faults lead to externally-observable failures. 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 do not correspond to broken dependencies.

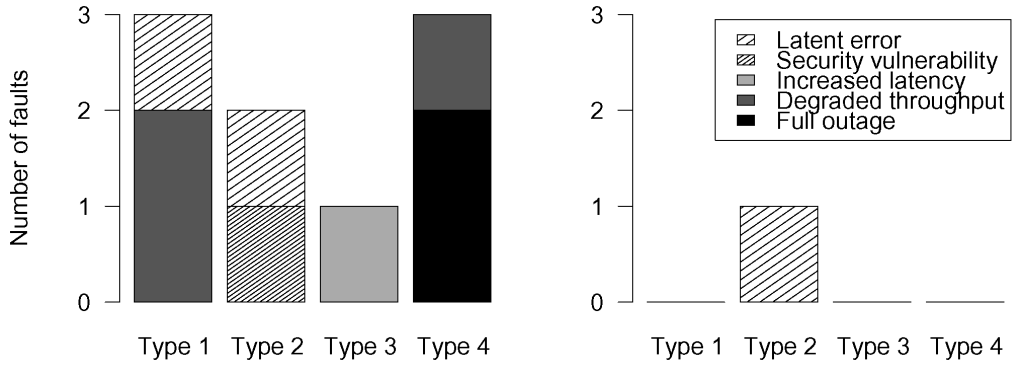

Figure 7. Impact of upgrade faults on a rolling upgrade(left) and on Imago (right).

This suggests that, while not eliminating all possible upgrade failures, Imago addresses the leading cause of such failures—breaking hidden dependencies. A statistical *t*-test shows that, compared with a rolling upgrade, Imago reduces the expected unavailability due to upgrade faults (result significant at the $p = 0.01$ level).

Conclusions and Future Work

Unlike previous research in online upgrades, I take a holistic approach and focus on upgrading distributed systems end-to-end, while guaranteeing three AIR properties (atomicity, isolation and runtime-testing). I identify the leading causes of planned and unplanned downtime, and I propose a novel upgrade-centric fault model. Through fault-injection experiments, I show that current approaches for upgrading distributed systems are unreliable because the upgrade is not an atomic operation and 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 complex data conversions, without requiring planned downtime, and allows testing the new version at runtime, in its operational environment. Imago trades resource overhead for an improved dependability of the upgrade. This overhead can be reduced by leasing resources from cloud-computing infrastructures or by relaxing the AIR properties and assessing the risk of mixed-version races. Because it avoids dependency tracking and states with mixed versions, Imago is likely to be easier to use correctly than previous approaches.

Existing online-upgrade approaches provide limited opportunities for testing the new version and the intermediate steps of the upgrade. Evaluating the dependability benefits of Imago's runtime-testing functionality represents an interesting avenue for future research. The efficient implementation of the AIR properties raises additional open questions, in particular for peer-to-peer systems, which accommodate large numbers of dynamically-added ingress points, or for data-intensive applications (e.g. MapReduce), which distribute their persistent data throughout the infrastructure and do not have a well-defined egress point.

References

S. Ajmani, B. Liskov and L. Shrira.
"Modular software upgrades for distributed systems."
In *European Conference on Object-Oriented Programming (ECOOP)*, Nantes, France, Jul 2006.

T. Bloom.
"Dynamic Module Replacement in a Distributed Programming System."
Ph.D. dissertation, MIT, 1983

F. Brooks.
"No silver bullet: essence and accidents of software engineering."
IEEE Computer, vol. 20, no. 4, pp. 10-19, Apr 1987.

A. Choi.
"Online application upgrade using edition-based redefinition."
In *ACM Workshop on Hot Topics in Software Upgrades (HotSWUp'09)*, Orlando, FL, Oct 2009.

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 (SOSP'07)*,
Stevenson, WA, Oct 2007.

C. Curino, H. Moon and C. Zaniolo.
"Graceful Database Schema Evolution: the PRISM Workbench."
In *International Conference on Very Large Data Bases (VLDB'08)*, Auckland, New Zealand, Aug 2008.

A. Downing, Oracle Corporation.
Personal communication, 2008.

T. Dumitraş and P. Narasimhan.
"Why do upgrades fail and what can we do about it? Toward dependable, online upgrades in enterprise systems."
In *ACM/IEEE/IFIP Middleware Conference (Middleware'09)*, Urbana-Champaign, IL, Nov/Dec 2009a.
Supplemental information: `http://www.ece.cmu.edu/~tdumitra/upgrade_faults/`.

T. Dumitraş and P. Narasimhan.
"No downtime for data conversions: Rethinking hot upgrades."
Technical Report CMU-PDL-09-106, Carnegie Mellon University, 2009b.

T. Dumitraş and P. Narasimhan.
"Upgrades-as-a-service in distributed systems."
In *USENIX File Conference on File and Storage Technologies (FAST'10)*, Work-in-progress reports, San Jose, CA, Jan 2010.

T. Dumitraş, P. Narasimhan and E. Tilevich.
"To upgrade or not to upgrade: Impact assessment for online upgrades of multi-tier enterprise applications."
In preparation, 2010.

F. Ferrandina , T. Meyer, R. Zicari, G. Ferran , J. Madec.
"Schema and Database Evolution in the O2 Object Database System."
In *International Conference on Very Large Data Bases (VLDB'95)*, Zürich, Switzerland, Sep 1995.

S. Hansell.
"Glitch makes teller machines take twice what they give."
The New York Times, Feb 1994.

Information Technology Infrastructure Library (ITIL) — Service Transition.
Office of Government Commerce, 2007

C. Koch.
"AT&T Wireless self-destructs."
CIO Magazine, Apr 2004.
`http://www.cio.com/archive/041504/wireless.html`.

J. Kramer and J. Magee.
"Dynamic configuration for distributed systems."
IEEE Transactions on Software Engineering, vol. 11, no. 4, Apr 1985.

D. Lowell, Y. Saito, E. Samberg.
"Devirtualizable virtual machines enabling general, single-node, online maintenance."
In *International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS'04)*, Boston, MA, Oct 2004

I. Neamtiu, M. Hicks, G. Stoyle and M. Oriol.
"Practical dynamic software updating for C."
In *ACM Conference on Programming Language Design and Implementation (PLDI'06)*, Ottawa, Canada, Jun 2006.

Oracle Corporation.
"Database rolling upgrade using Data Guard SQL Apply."
Maximum Availability Architecture White Paper, Dec 2008.

D. Reiss, Facebook.
Personal communication, 2009.

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.

I. Zolti, Accenture.
Personal communication, 2006.

Endnotes

[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.

[2] During the quiescence period required for switching over to the new version, the read-only client requests may be allowed to proceed.