Toward Upgrades-as-a-Service in Distributed Systems

Tudor Dumitraş
Carnegie Mellon University
tudor@cmu.edu

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

ABSTRACT

Unavailability in distributed enterprise systems is usually the result of planned events, such as upgrades, rather than failures. Major system upgrades entail complex data conversions that are difficult to perform on the fly, in the face of live workloads. Minimizing the downtime imposed by such conversions is a time-intensive and error-prone manual process. We propose upgrades-as-a-service, a novel approach that can eliminate all the causes of planned downtime recorded during the upgrade history of one of the ten most popular websites. Building on the lessons learned from past research on live upgrades in middleware systems, upgrades-as-a-service trade off a need for additional hardware resources during the upgrade for the ability to perform end-to-end upgrades online, with minimal application-specific knowledge.

1. INTRODUCTION

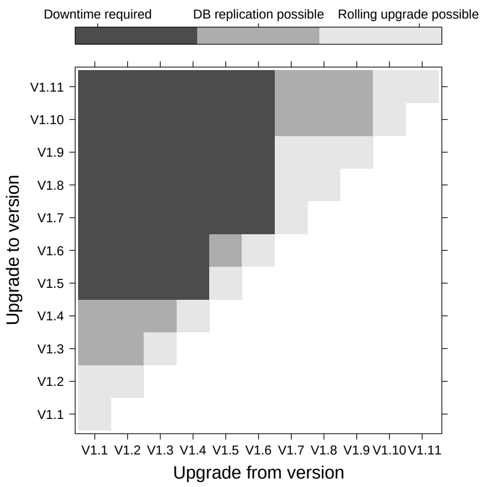
Software upgrades are unavoidable in enterprise systems. For example, business reasons sometimes mandate switching vendors; responding to customer expectations and conforming with government regulations can require new functionality. Moreover, many enterprises can no longer afford to incur the high cost of downtime and must perform such upgrades online, without stopping their systems. 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.

Previous research has concentrated on performing upgrades *in-place*, which requires tracking the complex dependencies of the system-under-upgrade, and on supporting *mixed versions*, which interact and synchronize their states in the presence of a live workload. The correctness of the mixed-version interactions is ensured through a time-intensive and error-prone manual process, *e.g.*, establishing constraints to prevent old code from accessing new data or conducting an in-depth pointer analysis to understand the nature and depth of the dependency chain. 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. However, 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.

2. CAUSES OF DOWNTIME

We study the upgrade history of Wikipedia—one of the ten most popular websites[1] to date—by combining data from a rigorous study of Wikipedia's schema evolution with information from design documents and archived discussions [3]. Wikipedia provides a multi-language, free encyclopedia, with 2.9 million articles. The content of these articles is stored in a 1 TB database, supported by 23 MySQL servers configured for master/slave replication. The master database receives the write queries and propagates the updates to the slaves, which handle read-only queries. The business logic of the system is a wiki engine called MediaWiki, which accesses the database and generates the content of the articles. Between 2003 and 2008, MediaWiki has released eleven versions (V1.1–V1.11), and the database schema has gone through 171 evolution steps [3].

When an upgrade of the wiki engine requires *schema changes*, which modify the definitions of the database tables, or *data conversions*, which modify the information stored in the database, Wikipedia tries to avoid downtime by performing a rolling upgrade. This technique removes database slaves one-by-one from the replication group, applies the schema changes, and then restarts the replication.The application servers are upgraded in a similar fashion, in a wave rolling through the data center. To support rolling upgrades, the database replication mechanism must allow source and target tables that do not have identical definitions. In general, however, rolling upgrades require the new version to be backwards-compatible. Only 5 out of the 55 possible upgrades among MediaWiki versions V1.1–V1.11 can be performed online, through a rolling upgrade [3]:



The history of Wikipedia upgrades [3] reveals five common reasons for upgrade-related downtime:

- Incompatible schema changes (*e.g.*, renaming tables in the database) prevent rolling upgrades and require upgrading the schema and the application in an atomic step;
- Data dependencies (*e.g.*, resulting from table joins) are hard to synchronize in response to updates issued by the live workload and can impose a high runtime overhead;

[1]According to http://www.alexa.com. Wikipedia handles peak loads of 70,000 HTTP requests/s.

- Long-running data conversions compete with the live workload and might overload the database;
- Bulk updates issued during such long-running conversions can interfere with the database replication;
- Competitive upgrades (*i.e.*, replacing the wiki engine with a different software) require data conversions and typically impose downtime.

Conversely, complex database changes are often avoided because they might impose downtime, even when this amounts to rejecting user-requested features. Furthermore, such complex upgrades often fail, *e.g.*, by breaking hidden dependencies in the system-under-upgrade [4], and cause unplanned downtime or data-loss.

3. UPGRADES-AS-A-SERVICE

We propose an alternative approach for software upgrades, which reduces the system unavailability by performing an offsite upgrade and by avoiding states with mixed versions. Relying on additional storage and computational resources—leased from existing cloud-computing infrastructures (*e.g.* the Amazon Web Services)—for the duration of the upgrade, we install the new version in a *parallel universe*, and we opportunistically transfer the persistent data from the production system in to the new version (Figure 1). This approach enables 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. As it effectively separates the mechanisms for performing an online upgrade from the functional aspects of the upgrade, we call this approach *upgrades-as-a-service.*

We have developped a prototype, called Imago [4], for performing online software upgrades with complex data conversions. The opportunistic data-conversion procedure takes into account the effect of the updates issued by the live workload and synchronizes the data in the new version. The business logic never interacts with a data schema belonging to a different version, and the live workload accesses either the old version (before the switchover) or the new version (after the switchover), but not both. Because the new version does not need to be in a consistent state until the switchover, the data dependencies are synchronized lazily, in the parallel universe, which prevents disrupting the live workload during the upgrade. Moreover, knowledge of the system's workload (*i.e.*, the database queries issued by the old and new versions of the business logic) can be incorporated in the conversion scheduler, in order to perform the incremental data-conversions efficiently. Imago supports all the schema changes that have imposed downtime during Wikipedia's upgrade history.

This approach can be implemented efficiently by using the GORDA API [1], which provides a uniform reflective interface for several database servers. Using this API, we monitor the data updates performed by the live workload by retrieving the object-sets from the executor stage of the old version's database. This reflective information allows us to avoid bypassing the database scheduler in order to control the serialization of concurrent transactions, a technique frequently used in systems for database clustering and replication (*e.g.* C-JDBC [2]). Aside from reducing the performance penalty during the upgrade, our approach allows us to test the new version in a realistic environment, where the concurrency-control mechanisms of the database are not disabled.

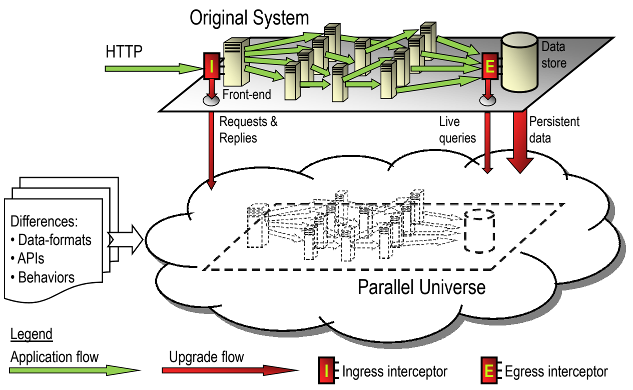


Figure 1: Upgrades-as-a-Service.

Current online-upgrades techniques, such as rolling upgrades, provide limited opportunities for testing the new version and the intermediate steps of the upgrade. Unlike these techniques, Imago does not upgrade the system in place. Imago's parallel universe allows us to test the new version online, using the live requests, before exposing its functionality to the clients. This testing strategy can reveal bugs and misconfigurations in the new version or incompatibilities with the deployment environment.

Additionally, we have shown that Imago significantly reduces the rate of upgrade failures by eliminating the risk of breaking hidden dependencies in the system-under-upgrade [4]. Therefore, *upgrades-as-a-service will reduce both the planned and the unplanned downtime* due to software upgrades. Because it does not require correctness constraints for the mixed-version interactions or knowledge of the old version's dependencies, this approach also reduces the manual interventions needed for preparing the upgrade and is easier to use than the existing techniques. From our experiences and observations with a previous online-upgrade approach [5], nearly a decade ago, we gained the intuition that leasing hardware resources for the duration of the upgrade costs less than the process of planning an in-place, online upgrade. Upgrades-as-a-service are likely to be more practically usable, less error-prone and better suited to fast upgrade cycles than the existing upgrade approaches.

4. REFERENCES

[1] 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*, 13(8):1110–1135, 2007.

[2] E. Cecchet, J. Marguerite, and W. Zwaenepoel. C-JDBC: Flexible database clustering middleware. In *USENIX Annual Technical Conference*, 2004.

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

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

[5] 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*, pages 184 – 196, Hilton Head, SC, Jan 2000.