No More *HotDep*endencies: Toward Dependency-Agnostic Online Upgrades in Distributed Systems

Tudor Dumitraş tudor@cmu.edu
Jiaqi Tan jiaqit@andrew.cmu.edu
Zhengheng Gho zgho@andrew.cmu.edu
Priya Narasimhan priya@cs.cmu.edu
Carnegie Mellon University
Pittsburgh, PA 15217

Abstract

Traditional approaches for online upgrades of distributed systems rely on dependency tracking to preserve system integrity during and after the upgrade. Because dependency reification can become intractable, we aim to enforce the isolation of the old and new versions during the upgrade. We achieve this by installing the new version in a "parallel universe" – a separate physical or virtual infrastructure that does not communicate directly with the old version. This allows our upgrading middleware to treat the complex IT infrastructure as a black box with unknown hidden-dependencies, and to validate the upgrade by cross-checking the outputs of the two universes.

1. Introduction

An *online upgrade* [1] is a change in the behavior, configuration, code, data or topology of a running application. Online upgrades have to consider the complex interactions between distributed components using specific APIs, networking protocols, queuing paths, configuration settings, etc. Such dependencies are not always well documented or understood, and are often hard to trace [2]. In general, complete dependency information cannot be automatically determined using either static analysis or runtime monitoring [2, 3].

An upgrading system must be careful not to disable existing applications (by breaking hidden dependencies), while still updating all of the components required by the new version being installed. Many online upgrades require massive amounts of data to be converted to new schemas, typically over a long period of time, and even as clients continue to perform transactions using the data under upgrade. Furthermore, an upgrade must be undoable, allowing administrators to roll back to the previous system if faults or unexpected behavior compromise the integrity of the upgrade.

We propose a *dependency-agnostic* online-upgrade approach that does not rely on dependency tracking and that does not induce downtime. Instead of an in-place upgrade, we aim to isolate the new version from the old. The new version, with a potentially different topology, is the result of a fresh installation and does not communicate directly with the old version. The old version's persistent data is transferred into the new version even as the old version continues to service requests. This is similar to the approaches used in single-host operating systems for isolating applications in virtual containers that prevent communication or cross-coupling between unrelated processes [4].

Our online-upgrade protocol avoids data staleness by invalidating the items that have changed during the data transfer. The old version is functional during the upgrade and remains intact afterwards. When the data transfer is complete, the two versions continue to run in parallel and to synchronize their states. As long as the versions run in parallel, administrators can cross-validate the outputs and roll back a failed upgrade.

This paper is organized as follows. In Section 2, we describe a case-study of upgrading a medium-to-large enterprise infrastructure. We discuss our dependency-agnostic upgrade protocol in Section 3 and its practical implications in Section 4.

2. Case study: Upgrading Wikipedia

To demonstrate our approach, we have chosen to mimic the medium-to-large infrastructure supporting www.wikipedia.org, a popular Web site providing a multi-language, free encyclopedia. Wikipedia has 5 million articles, which generate peak request rates of 30,000 HTTP requests/s (600 Mb/s incoming and 2.8 GB/s outgoing traffic). Each article receives 3 database-updates/s on average. This workload is supported by a multi-tiered infrastructure [5] with file servers and databases in the backend, running on 247 servers located in 4 data centers worldwide. The size of the database is 15 GB, not including images and other media files that are stored on the file servers.[1] The front-end has 52 caching proxies, accessed using round-robin DNS load-balancing. The proxies serve approximately 75% of the Wikipedia content, handling most of the page requests from visitors who are not logged in. The proxies forward the cache-misses to a load-balanced cluster of 150 web servers.

[1] These numbers are accurate as of Feb 2007, but Wikipedia grows at an exponential rate. For instance, in the English-language Wikipedia, the number of articles (currently 1.6 million) has doubled every 346 days. Wikipedia ran on 39 servers in 2005 and on 1 server in 2004.

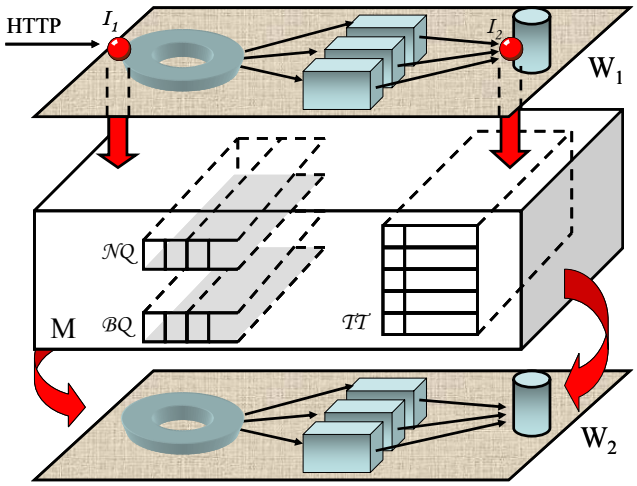

Figure 1. Dependency-agnostic upgrades. The old and new versions are installed and execute in parallel universes W_1 and W_2. The upgrading middleware M intercepts the request flow at the ingress (I_1) and egress (I_2) points of the old version. The rest of W_1 is treated as a black box.

The web servers generate the content of the pages using a wiki engine called MediaWiki [6], which is implemented as a set of PHP scripts. MediaWiki retrieves the text of an article from a database, running on 12 servers in a master-slave configuration, and the images and media files from a remote filesystem. The web servers also use PHP accelerators that cache compiled PHP scripts.

2.1. Wikipedia Dependencies

- *API dependencies*: Wikipedia relies on many shared libraries and third-party components. For instance, MediaWiki 1.9 requires PHP 5.0 and MySQL 5, while PHP requires Apache 1.3 or newer. There are also some optional dependencies: ImageMagick (itself dependent on third-party image manipulation libraries), a PHP accelerator for performance, etc. The Apache and MySQL daemons require a set of standard libraries, while PHP requires the MySQL client library. Some of these dependencies can be determined using static analysis, but others cannot (*e.g.* the web server loads the PHP interpreter library dynamically, triggered by a directive from a configuration file). Most of the upgrade-breaking API changes are due to refactorings (modifications of program structure, not intended to change its behavior) [2].
- *Configuration dependencies*: These are settings in the configuration files of MediaWiki and the other components that specify the available PHP accelerator, the path to the image directory, the PHP version, etc.
- *Protocol dependencies*: The front-end servers receive and handle HTTP requests, which may be forwarded to the servers in the middle tier. MediaWiki retrieves text from the database with SQL queries and image files from the filesystem with `read()` and `write()` system calls, while the MySQL clients connect to the database server using a binary protocol and the file servers provide the images using the NFS protocol.
- *Data dependencies*: In some cases, the behavior of the system cannot be determined from an HTTP request alone because it depends on the persistent data. For instance, if the text of an article contains a Math object, MediaWiki may invoke a LaTeX interpreter.
- *Performance dependencies*: The overall performance of Wikipedia depends on the software and hardware configuration. As the queuing paths in this infrastructure are very complex, performance issues that arise during an upgrade may be hard to diagnose. Moreover, the system behavior might depend on its performance: high latency can trigger communication exceptions; MediaWiki disables write-access to the database if the incoming load is too high, etc.

2.2. Upgrade Scenario

For upgrading to a new version, MediaWiki provides a script that inspects the database schema and converts it to the new format; this is a simple upgrade because it only involves the configuration files of the wiki software and the database layout — changes that can be made with the existing infrastructure left in place. Instead, we investigate a major and far more interesting upgrade scenario: switching to *a completely different wiki software*, such as TWiki [7]. While the two wiki engines (MediaWiki and TWiki) provide similar functionality, there are significant differences between them. The differences can be classified as *semantic* (e.g. TWiki has a fine-grained access control system; MediaWiki has a very detailed permission system, but no access-control lists); *behavioral* (deleting a page may have different outcomes, e.g. due to differences in the access control); *transmutability* (some data with identical semantics cannot be transferred between the two systems, *e.g.*, hashed passwords); *interface* (e.g. different URLs to access similar pages); *implementation* (e.g. TWiki stores its data in file system instead of a database); or *QoS* (throughput and response-time mismatches). This major-upgrade scenario (replacing a wiki engine and its dependencies) is realistic because switching vendors for business reasons is common in the IT industry.

3. Dependency-Agnostic Upgrades

The key idea behind our dependency agnostic upgrades is to install the new version in a "parallel universe" in order to isolate the old and new versions from each other. Figure 1 illustrates this technique. The original system W_1 has a parallel universe W_2 where the new version will run. W_1 continues to service incoming requests during the upgrade. The only communication channel between the two universes is via our upgrading middleware M, which continuously transfers the persistent data from W_1 to W_2, monitors the updates

```
PHASE I: BOOTSTRAPPING
    Initialize the transfer table TT with all the persistent data items to be transferred to W2; ∀x ∈ TT, TT(x) ← (invalid)
    Initialize non-blocking queue NQ for tracking in-progress updates and blocking queue BQ for enforcing quiescence
    Initialize interceptors I1 and I2

PHASE II: DATA TRANSFER
    while (∃x ∈ TT such that x was never transferred)
        x ← top(TT)
        Query x from the data store of W1
        Convert x to the data schema of W2
        Inject x into the data store of W2
        TT(x) ← (valid, transferred)
        Reorder TT such that top(TT) ∉ NQ and top(TT) is invalid
        if (I1 detects that data item y is updated) then
            NQ.enqueue(y)
        if (I2 detects that data item z is updated) then
            TT(z) ← (invalid)
            NQ.dequeue(z)

PHASE III: PARALLEL EXECUTION
    Stage 1: enforce quiescence
    Flush all caches from W1 and disable caching (or configure a write-through cache policy)
    while (NQ is not empty)
        if (I1 detects that data item y is updated) then
            BQ.enqueue(y)
        if (I2 detects that data item z is updated) then
            TT(z) ← (invalid)
            NQ.dequeue(z)
    Transfer all invalid items from TT

    Stage 2: Execute in parallel
    master_universe ← W1
    for all x ∈ BQ and all x intercepted at I1
        Send request(x) to both W1 and W2
        Propagate reply from master_universe to the client

PHASE IV: SWITCHOVER
    Discard volatile state (e.g. sessions)
    master_universe ← W2
    Continue with parallel execution
```

Figure 2. Pseudocode of the dependency-agnostic upgrade protocol.

handled by W_1 to prevent data-staleness and disables updates to W_1 to enforce quiescence.

For this purpose, we assume that the system has a few well-defined ingress and egress points. M transparently intercepts the request flow at the ingress points I_1, where the HTTP requests enter the old version, and at the egress points I_2, where persistent data is stored (e.g. the master database or the file system, in the case of Wikipedia). We use a transfer table *TT* to keep track of the transferred data items that have been updated, and a non-blocking queue *NQ* to monitor in-progress updates. The principal idea is that the information from I_1 and I_2 should be sufficient for maintaining data consistency, allowing us to treat the rest of the W_1 infrastructure as a black-box. Since the old version is rendered a black-box, all its complex dependencies end up being irrelevant to our upgrading process. I_1 also allows us to "lock down" the old version, using a blocking queue *BQ*, and to prevent W_1 from handling requests when the upgrade protocol requires a period of quiescence. Figure 2 shows the pseudocode of our protocol.

3.1. Protocol Phases

Bootstrapping. The biggest problem in bootstrapping the upgrade process is to capture in-progress updates, i.e. requests that trigger an article update and that have passed the ingress interception-point before the I_1 interceptor is operational but have not yet been committed to the database because they are still executing. This problem is aggravated by the presence of caches at various tiers in the infrastructure, which may delay the insertion of the update into the database.

In practice, since upgrades are often long-running processes, the in-progress updates will usually finish executing by the time W_2 is ready to start executing in parallel. To guarantee that no updates are overlooked, our upgrading middleware will flush all of the caches from the old system, or, as a last resort, restart the entire W_1 infrastructure, with the same effect.

Data Transfer. During this phase, we transfer the persistent data from W_1 to W_2, converting it to the new schema as needed. The mapping between the two schemas must be specified in advance, before starting the upgrade. Based on this mapping our middleware will attempt to find the closest equivalent of a data item in the new database. The main content of a wiki, represented by the text of the articles and the media files, can be accurately converted. Some items (e.g. links and certain statistics) do not need to be transferred because they can be recreated afresh. Others (e.g. formatting instructions for the wiki text) need to be transferred, but might have only an approximate equivalent in the new database. Finally, certain items cannot be converted at all, such as encrypted or hashed data (e.g. user-account passwords). Users must then reset their passwords when logging in to the new system for the first time. The transfer table *TT* keeps track of the data items already transferred. When the interceptor I_2 detects that a data item is updated in the old universe W_1, its corresponding entry in the page list is invalidated and the item is (re)scheduled for a fresh transfer to the new universe. The data transfer will eventually terminate if the transfer rate exceeds the rate at which previously converted data is invalidated.

Parallel Execution. After the database transfer is complete, the two universes may enter the parallel-execution stage. The middleware freezes the state of W_1 by blocking update requests (these are queued in *BQ* and applied later). When all of the outstanding updates have been committed to the database in the old version and transferred to W_2 (we determine this by comparing the requests observed at I_1 and I_2; the mapping between HTTP requests and database queries must be known in advance), the persistent states of the two universes are synchronized. W_1 and W_2 can start executing in parallel.

HTTP requests intercepted at I_1 are injected into both W_1 and W_2, after yet another conversion step. This step translates URLs in the old format for use with W_1 to the new, (approximately) equivalent form for use with W_2. Our upgrading middleware can then compare the two outputs in order to validate the upgrade's integrity. The mapping between corresponding URLs from W_1 and W_2 needs to be known in advance. Only the output from the master universe (W_1) is propagated to the clients.

Switchover. The switchover changes the master universe from W_1 to W_2. All new requests for URLs from W_1 will be automatically converted and redirected to W_2. Volatile state, such as user sessions, is discarded and users will be required to log in again. After the switchover, the two universes can continue to execute in parallel, allowing administrators to validate the upgrade by monitoring the outputs. The states of the two parallel will not be perfectly synchronized because of intrinsic behavioral differences between the two systems; indeed, this modified behavior could have been the very reason for initiating the upgrade. However, as long as the two universes continue to execute in parallel, our middleware can switch back and forth between W_1 and W_2. If the result of the upgrade is deemed inappropriate for some reason, the administrators can initiate a switchover from W_2 back to W_1, thereby rolling back the upgrade without loss of data.

4. Discussion and Conclusions

We propose a dependency-agnostic approach for performing major behavioral/semantic upgrades in complex distributed systems. This technique intentionally enforces isolation between the old and new versions by executing them in parallel universes and transferring data in the background. The parallel execution of the two versions provides a way to validate the upgrade by cross-checking their outputs. If needed, the upgrade can also be rolled back.

This approach assumes full knowledge of the mapping between the HTTP requests and database queries in both universes, and of correspondences between the requests in the two systems. In general, the behavior of the software needs to be well understood, as is the case for any upgrade strategy. Moreover, if the new version's parallel universe is virtual (e.g. realized via an overlay network), there may be performance dependencies between the old and the new versions.

The advantage of dependency-agnostic upgrades is that they allow us to ignore hidden dependencies between distributed components and to treat the entire IT infrastructure as a black box. The resulting upgrade is not a surgical procedure and is likely unsuitable for regular maintenance activities such as applying security patches. This approach is most appropriate for large-scale, major distributed upgrades because it avoids downtime and reduces the administrative burden by eliminating the need for dependency tracking.

References

[1] M. E. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, vol. 10, pp. 53-65, 1993.

[2] D. Dig and R. Johnson, "How do APIs evolve? A story of refactoring," *Journal of Software Maintenance and Evolution: Research and Practice*, vol. 18, pp. 83 - 107, 2006.

[3] F. Kon and R. H. Campbell, "Dependence Management in Component-Based Distributed Systems," *IEEE Concurrency*, vol. 8, pp. 26-36, 2000.

[4] S. Potter and J. Nieh, "Reducing Downtime Due to System Maintenance and Upgrades," in *LISA*, San Diego, CA, 2005, pp. 47-62.

[5] https://wikitech.leuksman.com/view/Server_roles.

[6] MediaWiki, http://www.mediawiki.org/wiki/MediaWiki.

[7] TWiki, http://twiki.org/.