MetaMorphMagi:
From Offline to Online Software Upgrades in Large-Scale IT Infrastructures

Tudor Dumitraş Jiaqi Tan Priya Narasimhan

June 20, 2007
CMU-CyLab-07-008

CyLab
Carnegie Mellon University
Pittsburgh, PA 15213

MetaMorphMagi:
From Offline to Online Software Upgrades in Large-Scale IT Infrastructures

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

Abstract

Software upgrades are one of the leading causes of downtime in IT infrastructures. Long running data-migration processes require intensive up-front preparation, extended maintenance windows and close monitoring, and they impose a significant burden on the system administrators. Even worse, major upgrades sometimes fail due to complex, hidden dependencies within the system, causing unplanned downtime and loss of critical data. In this paper, we propose a technique for converting an offline data-migration process into a dependency-agnostic online upgrade that requires minimal administrative intervention and that eliminates the need for planned downtime. We illustrate our technique by walking the reader through a hypothetical, but realistic online upgrade scenario in a medium-sized IT infrastructure – namely, hot-swapping the wiki software that underlies Wikipedia with an entirely different wiki engine.

Prologue

In November 2003, the upgrade of a customer relationship management (CRM) system at AT&T Wireless backfired, causing chronic downtime in several key systems, which affected thousands of customers. The CRM system depended on 15 different legacy systems, and the federally-mandated Nov 24th deadline for implementing wireless-number portability required the system to be integrated with the IT systems of other companies. The complex dependencies of the CRM system on various legacy back-ends became unmanageable, and the integration proved too difficult to test in a realistic offline environment. As the deadline approached, IT managers considered rolling back to the old CRM software; however, they had not preserved enough of that previous system, and were forced to go forward with the upgrade. When the first number-porting requests arrived, the system broke down, creating a ripple effect that disabled other AT&T systems. Number-portability requests were not handled automatically, error messages were not recorded (making post-upgrade manual resolution difficult), the CRM system was down and customer-service representatives were unable to help the estimated 50,000 new customers per week who were trying to activate their service. The negative side-effects of the upgrade persisted for 3 months. All said and done, the failed upgrade cost AT&T Wireless $100 million and damaged the company's reputation, with dire consequences for the future of the company [1].

The misadventure of AT&T Wireless teaches us two important lessons about software upgrades. The first lesson is that major upgrades are hard and risky. Industry analysts indicate that "an average of 80 percent of mission-critical application service downtime is directly caused by people or process failures," which are usually related to change management [2]. A study conducted in 1998 at 426 sites found that planned outages for performing change management operations accounted for 75% of 5921 outages and that this fraction increases over time [3]. The ability to perform such changes online, with no downtime and minimal human intervention, ranks high on the wish list of IT administrators and service providers. An *online upgrade* is a change in the behavior, configuration, code, data or topology of a running system [4]. Online upgrades form an essential building block for achieving the holy grail of self-regulating, autonomic management of large enterprise systems.

The second lesson is that dependencies are hard to manage in complex IT infrastructures. Every user of modern operating systems is painfully familiar with a phenomenon colloquially known as "DLL Hell" [5]. This involves the user having to disentangle and decipher the dependencies between shared libraries, upon the installation or upgrade of any application. These dependencies are not always well documented, and they are very hard to track. In distributed systems, there emerge additional sources of dependencies due to the applications' reliance on specific networking protocols, middleware, routing paths or performance levels. In general, complete dependency information cannot be detected automatically [6, 7]. We have previously introduced a protocol, not based on dependency tracking, for performing online upgrades with no data loss and minimal downtime [8].

In this paper, we present MetaMorphMagi (M^3), a technique for converting a standard offline-upgrading process into a dependency-agnostic online upgrade. We target *major* upgrades of large-scale enterprise systems, where we replace an entire IT infrastructure with a new system (or with a new version that is substantially different from the old one) that might exhibit differences in topology, dependency tree, data layout and performance profile. M^3 can convert an offline data-migration process into a long-running, but fully automated, online upgrade. This technique can be applied if four conditions are satisfied:

- We must be able to *stop* the data-migration process *and* resume from where we left off. This is also the minimal requirement for recovering from failures that interrupt the data migration.
- We must be able to *monitor the flow of requests* and analyze their effect on the system's persistent data. This helps us to avoid data staleness by re-transferring any updated items on the fly.
- We must be able to *read* the data from the old version *without disrupting* the functionality of any running applications using the old version. This requirement means that we cannot lock down any database tables or overload running servers.
- We must be able to *"lock" the system for writing* (read queries may still go through) and flush all the in-progress updates. This allows us to enforce a brief period of quiescence before switching over to the new version.

We avoid the problem of tracking dependencies by isolating the new version from the old one. We install the new version in a "parallel universe" – a separate physical or virtual infrastructure that cannot communicate directly with the old version. Since this is a fresh installation, the links between the new system's components are created through the usual installation process rather than by attempting to reproduce and maintain the dependencies from the old system. The old system is functional during the upgrade and remains intact afterwards, allowing the administrators to roll back the upgrade if necessary. We trickle the persistent data into the new system's data store while the old system is running and servicing requests.

The primary contribution of this paper is a novel technique for converting an offline data-migration process into an online upgrade counter-part. The resulting upgrading process does not rely on a complete knowledge of the dependencies from the IT infrastructure and it uses the protocol introduced in [8]. Our goal in developing and presenting this approach is to leverage and complement existing best practices for major upgrades of IT infrastructures, instead of substituting them with an incompatible process that re-invents the wheel.

For the sake of clarity, we illustrate each step of our conversion technique with an example taken from a case study that considers a credible, albeit hypothetical, upgrade of a well-known IT infrastructure. Wikipedia is a medium-sized (around 250 machines), three-tiered IT infrastructure that supports multiple services and relies on a couple of data-stores in the backend [9]. We address the problem of upgrading the software underlying Wikipedia [10] to a radically different wiki engine [11]. We explain the fundamental differences between these two systems (for instance, the new wiki engine stores its persistent data in a flat filesystem rather than in a database), and we show how to perform the data migration through an online upgrade.

This is a story that unfolds in five acts: in Act 1 we present the upgrade problem by describing the Wikipedia infrastructure and data, as well as the major differences between the old and new wiki engines. In Act 2 we analyze the data migration process required for an offline upgrade. In Act 3 we show the additional steps needed for performing the upgrade online, while the old infrastructure is still servicing requests. In Act 4 we discuss the requirements for running the old and new infrastructures in parallel, in order to cross-validate the upgrade, and in Act 5 we survey related approaches. We conclude by summarizing our main contributions.

1 From M to T: the Wikipedia Adventure

Wikipedia (www.wikipedia.org) is 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 per second (500 Mb/s incoming and 3 Gb/s outgoing traffic). Each article is edited 22 times on average over its lifetime. This generates about 3 database updates each second. This workload is supported by a multi-tiered infrastructure with file servers and databases in the backend, running on 253 servers located in 4 data centers worldwide. The current size of the database is 15 GB, not including images and other media files that are stored on a distributed filesystem.[1]

Figure 1 shows the Wikipedia infrastructure [9]. The front-end has 67 caching proxies (running Squid [12]), which are accessed using round-robin DNS. The proxies serve approximately 75% of the Wikipedia

[1] These numbers are accurate as of May 2007, but Wikipedia grows at an exponential rate. For instance, in the English-language Wikipedia, the number of articles (currently 1.8 million) has doubled every 346 days. Wikipedia used 39 servers in 2005 and 1 server in 2004.

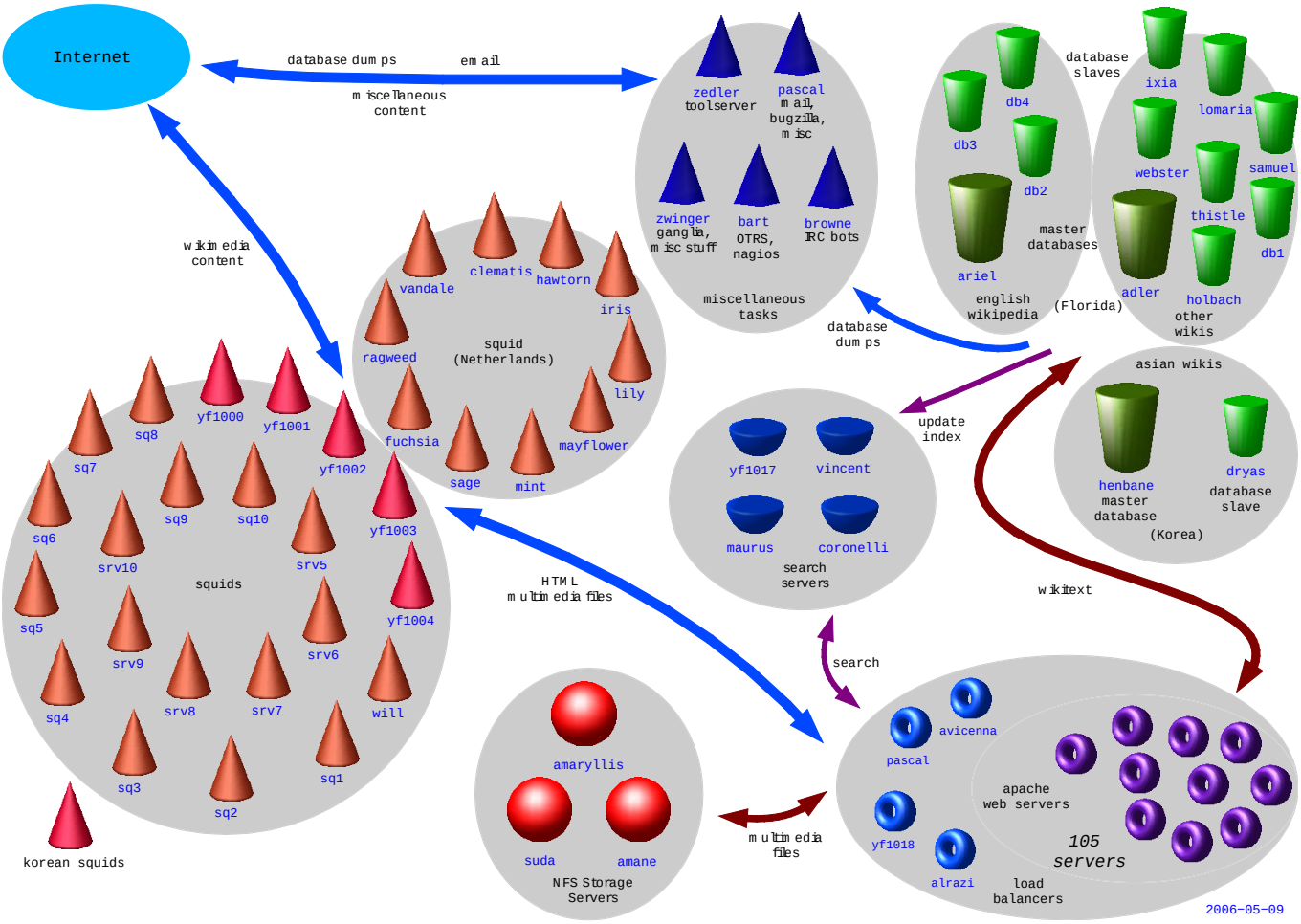

Figure 1. The Wikipedia infrastructure in May 2006 (from *http://en.wikipedia.org/wiki/Wikipedia*). In May 2007 the infrastructure has a similar architecture, but the number of servers has increased by 64% to sustain a 100% increase in the incoming load.

content, handling most of the page requests made by visitors who are not logged in. The proxy cache-misses are forwarded to a cluster of 150 Apache web servers [13], load-balanced using LVS [14].

The web servers generate the content of the pages using a wiki engine called MediaWiki [10], which is implemented as a set of PHP scripts. MediaWiki retrieves the text of an article from a MySQL database, running on 18 servers in a master-slave configuration, and the images and media files from a remote NFS filesystem. The web servers also use PHP accelerators that cache compiled PHP scripts. While most requests come from external search engines such as Google, Wikipedia also has 18 load-balanced search servers that run the Lucene indexing and search software [15].

For upgrading to a new version of itself, MediaWiki provides a script that inspects the database schema and converts it to the new format; this is a relatively simple upgrade because it only involves the configuration files of the MediaWiki software and the database layout. Instead, we investigate a *major* upgrade scenario, namely, switching to *completely different* wiki engine, TWiki [11]. TWiki is implemented in Perl, it uses a different data layout and it does not rely on a database. Instead, TWiki stores all of its persistent data on a filesystem with RCS versioning [16]. While upgrading this infrastructure, we also re-design the entire multi-tiered architecture to optimize it for the new wiki engine. While this scenario is not entirely realistic for our example (the MediaWiki software is developed specifically for Wikipedia), it is common in the IT industry to upgrade when business reasons dictate a switch to a different vendor.

1.1 A Tale of Two Wikis: M(edia) & T Wiki

Both MediaWiki (MW) and TWiki (TW) provide similar wiki functionality. Both systems allow visitors to view and edit the content of articles, called "pages" in MW and "topics" in TW, which are organized in "namespaces" in MW and "webs" in TW. There are, however, considerable differences between the two wikis stemming from the fact that they target different user groups. MW is designed to support the world's largest encyclopedia, with the goal of allowing a large number of users to access and modify content concurrently (according to the statistics from alexa.com, Wikipedia is one of the 10 most popular sites on the Web). TW aims to support enterprise collaboration platforms by providing structured content in a corporate setting and by allowing administrators to create customized applications based on the wiki engine.

We have classified the differences between MW and TW into six categories: semantic, behavioral, transmutability, interface, implementation and quality-of-service (QoS). These categories are relevant for different steps of the upgrade: the offline migration process handles the semantic differences, QoS and implementation differences are important during an online upgrade, interface and transmutability differences have to be handled when switching over to the new system, while behavioral differences become relevant when trying to run the new and the old system in parallel for validating the upgrade.

1.1.1 Semantic differences refer to persistent *data* (or meta-data) that has different meanings in the two wikis. Typically, this would be a data item from one wiki that has no semantic equivalent in the other one. Examples of semantic differences between MW and TW are:

- *Access-control*. TWiki provides a fine-grained access-control system, with access-control lists (ACLs) at the system, web and topic levels, while MediaWiki has more detailed permissions (*e.g.* `createtalk` or `undelete`) that apply system-wide and that require modifications to the configuration files and the database.
- *Talk pages*. In MediaWiki, each regular page has a talk page associated, where users can discuss the subject of the article. There is no semantic equivalent in TWiki. In practice, TWiki users create a "Discussion" section at the bottom of a page that serves the same purpose, but there is no way to specify that the content of this section has a special meaning.
- *Logging the reason for edits.* The users of MW can record the reason for creating a new revision, while TW doesn't maintain such metadata.
- *Hierarchical structure.* Each topic in TW has a `TOPICPARENT`, creating a hierarchical structure. MW pages are organized in flat namespaces.

1.1.2 Behavioral differences refer to similar *actions* that lead to different outcomes in the two wikis. These differences are visible to the users of the wikis. Semantic differences may induce behavioral differences: for instance, due to the dissimilar access control schemes, a request to edit a page can have different outcomes in MW and TW. Moreover, many configuration settings affect the behavior of the wikis, creating behavioral differences. Examples of behavioral differences between MW and TW are:

- *Searching for a page*. If a page with the requested title is not found, MW displays the Web form to start editing that page (although this behavior is configurable). TW presents a message informing the user that the page does not exist.
- *Undeleting pages*. In TW, deleted pages are stored in the Trash web and can be restored together with all their revisions. In MW, a deleted page can be restored by moving it out of the `archive` table, but it is impossible to cleanly restore the entire page history because the old page id connecting all the revisions is not recorded.
- *Anonymous edits*. By default, MW allows anonymous users to edit pages (recording the IP address of the user as the source of the edit). TW requires users to be loged in before editing anything.

1.1.3 Transmutability differences refer to equivalent data items in the two wikis, but that cannot be converted form one format to the other. For example, passwords encoded through a one-way hash function cannot be re-encoded or converted to a different format.

1.1.4 Interface differences refer to actions or data that have equivalent or similar semantics in the two wikis, but that are accessed through different names or APIs. Examples of interface differences between MW and TW are:

- *URLs*. The paths and URLs used to retrieve articles are different in MW and TW.
- *WikiWords*. In MW, users and articles may have arbitrary names; in TW, all these names must be WikiWords (words that contain at least two capital letters and no spaces, *e.g.* WikiWord).

1.1.5 Implementation differences refer to similar functionality implemented in different ways in the two wikis. MW and TW are implemented in different programming languages (PHP and Perl, respectively), and the implementations have very little in common (one exception from this is that both wikis ultimately

rely on the GNU diffutils for comparing page revisions). Some of these differences are relevant for the upgrade process, for example:

- *Datastore.* MW uses a database and a filesystem in the backend, while TW uses a filesystem with RCS versioning.
- *Page Histories.* MW stores the entire text of all the past revisions (compressed in some cases), while TWiki stores them as reverse diffs (provided by RCS).

1.1.6 QoS differences refer to throughput and response-time disparities between the two wikis. These are heavily dependent on the software and hardware configuration: configuration settings for the wikis, web servers, databases, the use of PHP accelerators and Perl in-memory interpreters, caching, memory available and CPU speed, etc. While the differences from the first five categories are due to the design of the two wiki engines, QoS differences also derive from the properties of the corresponding IT infrastructures.

2 Offline Upgrade: On the Care and Feeding of a Baby Encyclopædia

The key part of any upgrade is transferring and converting the persistent state of the application to a format that the new version can understand. We start by installing TWiki on a new infrastructure, isolated from the original Wikipedia, and by configuring and tuning this installation to achieve the desired performance characteristics. Techniques for improving the performance of IT infrastructures are covered elsewhere in the system administration literature [17]; for keeping this presentation focused on upgrades, we concentrate on the details of migrating the existing Wikipedia system to a TWiki-centered infrastructure. We have developed a system called MetaMorphMagi to demonstrate how such an intricate upgrade can be accomplished, and how it can be performed online.

At this point, we have a well oiled, spanking new version of the infrastructure without any content; the next step is to feed all the Wikipedia data into this young encyclopedia. This data migration step must reconcile all the semantic differences between the two systems. The two wiki engines are comparable in size: MW has about 70 KLOC of PHP code (not including the maintenance scripts), and TW has 60 KLOC of Perl code. By reading the online documentation, examining the source code and experimenting with the software, we have created a mapping between the data items of MW and TW. The effort to produce this mapping has required 72 man-hours of work. The mapping is summarized in Figure 2 and is presented in detail in the Appendix.

We have realized early on that the tables from the MW database do not have exact equivalents in the data layout of TW. The MW data items have many-to-many relationships to TW data. For instance, the field `user_id` from the MW `user` table determines the user name belonging a user group, responsible for an entry recorded in the `logging` table or having authored an article revision stored in the `revision` table; these user names correspond to the names stored in the user groups, the statistics files and the RCS revisions on the TW side. Conversely, to create a TW revision we need information from several MW database tables (*i.e.* the `page`, `revision`, `text` and `user` tables).

Because of these complex relationships between MW and TW, we have to define a set of logical items to drive the data migration. The migration of one logical item should be atomic (*i.e.* the item must be transferred entirely or not at all). These basic logical items cannot be the rows of the MW database tables because the tables usually contain foreign keys and the migration will involve multiple tables. We cannot transfer page-by-page either because one page corresponds to a lot of data (multiple revisions, images and image revisions, logs, user who edited the page, etc.), and rolling back a transfer at this coarse level of granularity would mean throwing away and redoing a great amount of work.

After examining the data items with similar semantics in MW and TW, we have identified eight categories of logical items to migrate: users, user-groups, namespaces, pages, revisions, archives, notifications and statistics. Transferring a page-item means creating a TW file with the appropriate name (the title of the article) and the corresponding permissions. The actual article content is transferred with the revision-items. The archives contain all the revisions of the deleted items and they correspond to the `Trash` web from TW. This division of logical items also imposes a logical order of migration: a revision cannot be transferred before the page to which it belongs. Formally, this is a partial order because some items are equivalent and can be transferred in any order.

We have designed the data-migration component of M^3 with the goal of distributing the conversions on several machines, in order to speed up the process. The migration component has a migration driver, which schedules the conversion of each data item from Wikipedia, and a stateless conversion library. The library contains routines for converting each type of logical items from MW to a TW format. Each conversion is atomic: the results are not permanent until the driver decides to commit them to the destination (TW) data store.

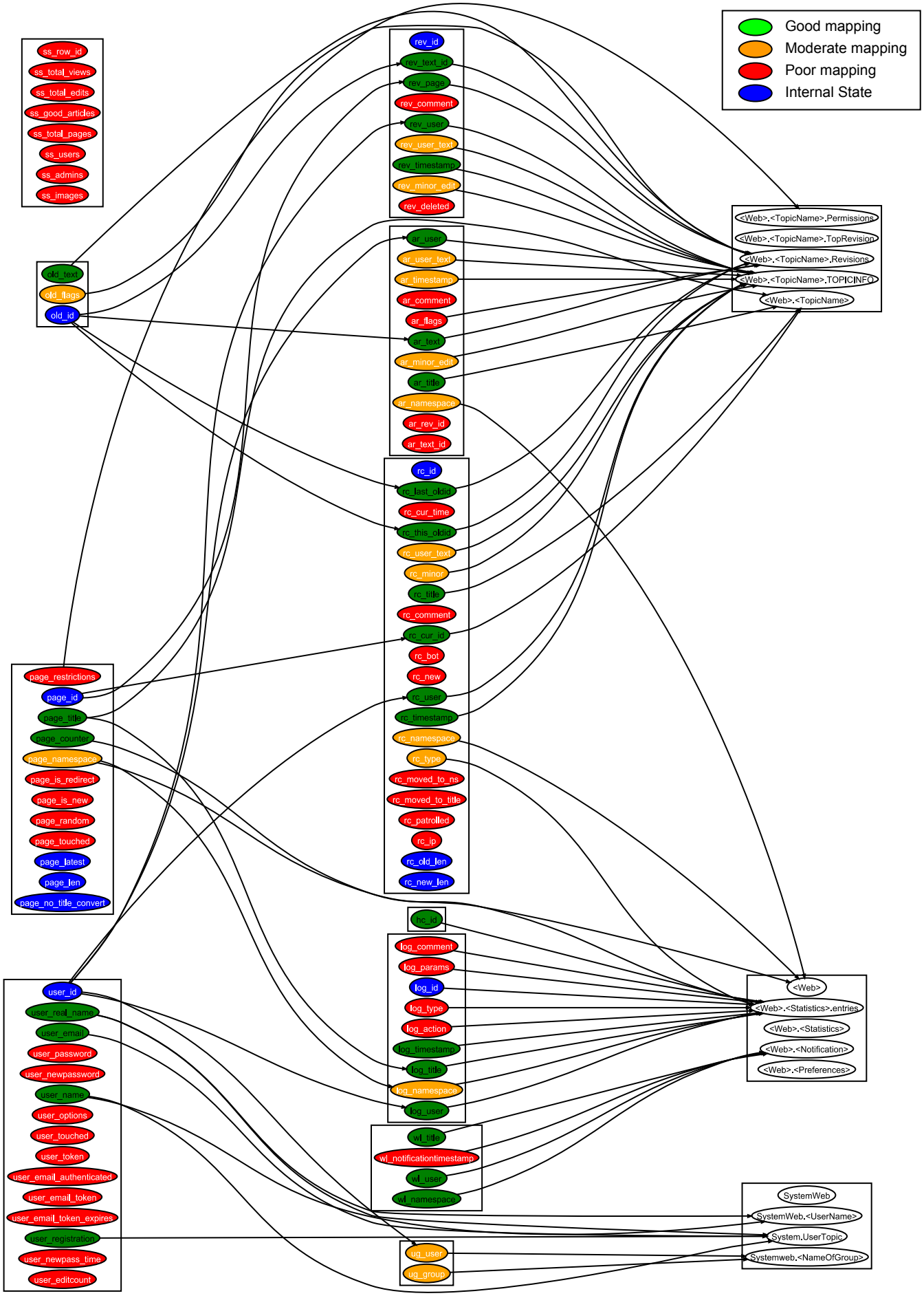

Figure 2. Mapping between primary keys in MediaWiki (left), other data items in MediaWiki (center) and data items in TWiki (right).

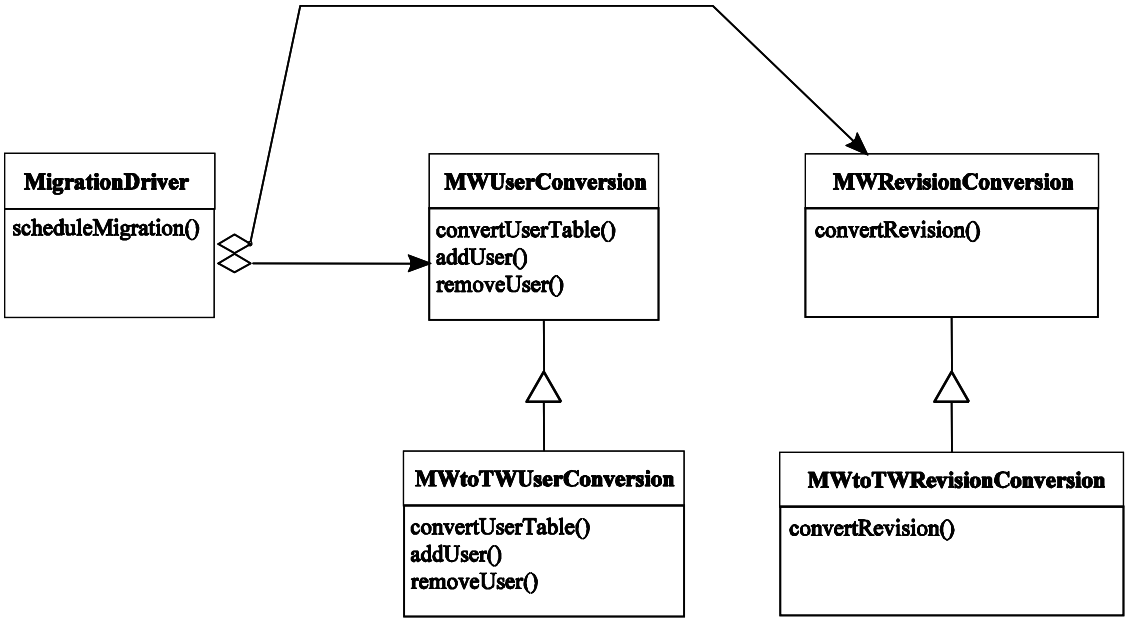

Figure 3. Data-migration component of MetaMorphMagi. The migration driver parses the MediaWiki database and schedules the conversion of each logical data item (e.g. users, pages, revisions).The conversion interface is defined in base classes such as `MWUserConversion` and `MWRevisionConversion`, which contain empty implementations of all the conversion methods. These empty methods are overridden in subclasses such as `MWtoTWUserConversion` and `MWtoTWRevisionConversion`, which implement the concrete conversions to the TWiki data format.

After a successful conversion, the library routine saves the results in a temporary file. The driver uses these files to gradually build the TW persistent data store. When the conversion fails, the library routine throws an informative exception, and the driver determines the appropriate way to handle this failure (*e.g.* reschedule the conversion at a later time or pester a system administrator). As the migration of each data item is atomic and the conversion library does not maintain any state, running multiple conversions in parallel is very straightforward. These parallel conversions may execute on the same machine or on a remote cluster. The driver is a centralized process that schedules all the conversions taking into account the logical order among items and that manages the farm of "migration workers".

We use a "parse once, convert to anything" pattern to separate the parsing of the original data from the actual data conversion (see Figure 3). The interface for converting each logical item is defined in a base class, while a concrete subclass contains the implementation. The concrete subclass may define additional methods, including helper functions that may be useful to the other concrete conversion routines (*e.g.* converting a string to a WikiWord). The driver is programmed to use the interface provided by the base classes; however, the objects that the driver uses are created as instances of the concrete subclasses. The base classes are not abstract; instead, they contain empty implementations of all the conversion methods. The concrete subclasses override these empty implementations. This design separates the parsing of the MW data from the conversion implementation and allows us to reuse the driver, with minimal porting effort, for converting from MW to another wiki engine different than TW. If a particular data item does not have an equivalent in the new wiki, the corresponding base class will not be overridden. In this case, the driver will invoke the empty implementation from the base class and the conversion schedule will continue with the other logical items.

To clarify our implementation, we present in detail a few conversion examples. The main content of the encyclopedia (the text of all the articles) is migrated with the page and revision items. A MW page is uniquely identified in the `page` table by a primary key called `page_id`. MW uses this key to find all the revisions of the page in the `revision` table. Each revision has a `rev_text_id` that points to the actual wiki text of the revision from the text table (see Figure 2). We use a similar approach in the data migration process. We identify the current name of the page corresponding to a `page_id` and we invoke the conversion of this data item. The data conversion

library transforms this title into a WikiWord (by capitalizing the first letter of each word and removing all the spaces) and returns it to the driver. This will be the name of the file containing the most recent revision of the article in TW. The driver then invokes the conversion of all the revisions of this article, in the order in which they were created. TW uses RCS versioning to store the revisions; this means that the most recent version of the article is stored uncompressed, while the older versions are stored in a separate file as incremental reverse diffs from the current revision. We leverage RCS for adding revisions in the same way as TW does during normal operation. The conversion library receives the new revision to check in and the two RCS files corresponding to the previous revisions. The library converts the MW-specific wiki syntax into TW syntax, then invokes RCS to add the new revision and returns the resulting temporary files. Upon successful completion of this conversion the driver copies the new RCS files to their permanent destination.

In general, this approach for incremental migration works well for logical items where data can be appended in the TW file formats. Migrating users presents an exception from this assumption because the TW user list is sorted in alphabetical order. This means that we cannot add an arbitrary user without recreating the entire file. For this reason, we define a method `convertUserTable()` that migrates all the users in one shot. The conversion library is still stateless; the user migration is a long-running operation that may fail and may need to be restarted, but the classes that perform the user migration do not keep any state in-between invocations. This conversion routine transforms the user names into WikiWords using the same helper function involved in the page conversion.

While we can define exact conversions for pages, revisions and user lists, migrating archived (deleted) pages is a best-effort conversion. When deleting a page, MW moves the text and titles of all the revisions in the `archive` table, but it does not save the old `page_id`. This means that there is no clean way of recovering the entire history of a deleted page for placing it in TW's `Trash` web. We apply a heuristic that compares the titles of all the revisions and converts them in the order of their timestamps; however, multiple title changes in MW may prevent this algorithm from accurately identifying all the revisions that belong to the same page.

3 Online Upgrade: Taming of the Slew

Wikipedia receives up to 30,000 requests per second. When performing an online upgrade, the data migration process competes with this slew of requests for accessing the persistent data. The online upgrade must be carefully executed to avoid perturbing the performance of the online system, to circumvent the problem of hidden dependencies, to preserve the overall system correctness and the data integrity and to prevent data staleness. We have previously proposed a protocol for performing upgrades in a dependency-agnostic manner [8]. In this section, we show how we can adapt an offline data migration process to use this online protocol, and we apply this technique to our Wikipedia upgrade example.

3.1 Prerequisites for Converting the Offline Migration to an Online Upgrade

Offline data migration processes, such as the one presented in Section 2, implement the core functions of an online upgrade: (i) installing and configuring the new system, and (ii) transferring the persistent data from the old system. When implementing an online upgrading system we can reuse the code developed for offline data migration almost entirely. There are four prerequisites for successfully building an online upgrader on top of an offline data migration process: fault-recovery capabilities, the ability to intercept and monitor the request flow, avoiding interference with the online system while accessing the data, and the ability to disable write access to the online system for a brief window of time.

3.1.1 Fault recovery. We must be able to restart the data migration after an interruption, such as the one caused by a hardware crash. Fault-recovery capabilities greatly simplify the upgrade because they allow us to save multiple checkpoints marking the progress of the data migration and to restart certain transfers if needed.

The design we have chosen for the data-migration component of M^3 allows us to add the fault-recovery functionality with very little effort. The data conversion library is stateless (see Figure 3) and all the conversion routines are atomic. The conversion routines create temporary files that the driver copies to the TWiki data directory. As a result, any conversion can be safely restarted if the conversion process fails before completing the migration. The migration driver maintains a persistent list of logical data items to convert; this list is saved after each successful completion of a logical item migration. We can stop the driver at any time and restart it later. The migration will continue from the point where it was interrupted. Moreover, we can add new items to the transfer list or mark some entries as dirty and schedule them for retransfer in order to account for the activity of the online system.

3.1.2 Request interception. During an online upgrade, the live system may update its persistent datastore by adding new items or modifying existing ones. To avoid data staleness, we need to monitor the request flow and to update the transfer list accordingly. In multi-tier IT infrastructures, such as the one supporting Wikipedia (see Figure 1), data updates may take a long time to reach the backend datastore due to network and processing delays and to write-caching at multiple tiers in the infrastructure. We need to monitor these in-progress updates by intercepting the requests at the ingress (where incoming requests enter the infrastructure, *e.g.* the URL of the main Wikipedia page) and egress (where the persistent data is stored, *e.g.*, the backend database) points of the infrastructure.

Interception must be transparent to the online application and it must not affect its performance. While we may stop and restart the data migration at any time, the interceptors must be always on. If we miss some in-progress updates due to interceptor unavailability, we will have incomplete information about data staleness and we will be forced to restart the migration from the beginning.

Information from the ingress and egress points is sufficient for maintaining data consistency during the online upgrade. The egress-point interceptors allow us to monitor the database updates and to schedule all these updates for transfer in the migration driver. The difference between the updates seen at the ingress and egress interceptors represents the in-progress updates, which have entered the infrastructure but are not yet reflected in the database. In order to compute this difference, the behavior of the software must be well understood, and the mapping between HTTP requests and database queries must be known in advance.

There are many techniques that can be used to intercept the request flow at the ingress and egress points. We review four interception techniques that we have considered for our Wikipedia-upgrade case study: network sniffing, log-file analysis, library interposition and round-robin request tunneling. The most appropriate combination of these techniques for implementing the ingress/egress interceptors depends on the application characteristics and the infrastructure configuration. We present each of these mechanisms, and we explain the contexts where they can be used successfully.

Network sniffing is perhaps the simplest approach for monitoring the request flow. Many IT infrastructures monitor their network usage using packet sniffing. This functionality allows us to observe the incoming requests entering the infrastructure, as well as the requests for the database or file servers in the backend. However, there are situations where sniffing cannot capture the entire request flow. For instance, requests using secure connections (*e.g.* for `https://` URLs) are encrypted and their content cannot be analyzed. In some cases, the server and the client sending the request reside on the same host (*e.g.* when using a slave database server); since these requests do not traverse the network, they are not visible to the network sniffer.

We can overcome these disadvantages by using *log-file analysis* instead. By parsing, sorting and correlating the entries from the log files of the web and database servers, we can retrieve encrypted requests and requests not visible to a network sniffer. Due to storage concerns, however, request logging is often turned off or the entire content of the requests is not recorded. Configuring the servers to log all the data we need usually requires restarting server daemons, which induces a brief downtime. More importantly, we need to analyze the logs from every single server in the infrastructure; if we miss one server, some requests will go by unintercepted. Therefore log-file analysis is probably unsuitable for websites with high incoming request rates, or where the configuration of the servers changes frequently.

These passive interception techniques do not introduce any coupling between the request monitoring and the normal processing. They cannot block requests for the live system and have no impact on its performance. This is a desirable property, but it also has a hidden disadvantage: because the request flow is not controlled in any way by the interceptors, it is easy to overlook requests that manage to bypass the interception system. The other two interception mechanisms that we present here place the interceptors on the queuing paths of the requests, which allows them to throttle the request rates and even to block certain requests, if needed (*e.g.* when disabling write access to the online system).

Library interposition [18] allows us to intercept the system calls of the server processes. The interceptor is a shared library that redefines the standard system calls, such as `read()` and `write()`. These redefined calls are interposed between the application and the system libraries, such that, at runtime, the application (unknowingly) calls the functions from the distributor, rather than the standard ones. Library interposition offers the great advantage that it is always on when the corresponding server is on, as the interceptor is part of the same process as the server. The interceptor stops working only when the server is down, ensuring that we do not miss any requests. Interceptors based on library interposition have similar disadvantages as log-file analysis: they require the server daemons to be restarted (library interceptors cannot usually be attached on the

fly), and they need to be attached to all the servers in the infrastructure.

Round-robin request tunneling is most appropriate for ingress-point interceptors in infrastructures using multiple front-end servers. For example, Wikipedia has 52 caching web proxies in the front-end; a DNS load-balancer assigns the domain name www.wikipedia.org to the IP addresses belonging to one of the front-ends, cycling through these proxies in a round-robin fashion. We intercept the DNS query using library interposition and we modify the response by pointing it to the IP address of a special-purpose web proxy that logs all the incoming requests and forwards them to the original front-ends. As all the requests pass through this new web proxy, we risk introducing a bottleneck for the entire infrastructure. We therefore spread the load over multiple interception proxies, and we use direct the requests to each of these proxies using round-robin DNS. The original front-end servers where the requests are tunneled are also selected following a round-robin scheme. We do not need to synchronize the logs of the interceptor proxies because the order of receiving the requests is not important; the task of the interceptors is to monitor the in-progress updates. Round-robin request tunneling does not rely on any knowledge of the infrastructure configuration. Instead, it inserts interceptors in the spot that all the requests use as the unique entry point of the system: the URL of the main Wikipedia page.

3.1.3 Interference avoidance. The request interception and data migration must not alter the performance or the functionality of the online system. The interceptors and the additional load on the database servers due to the migration processes introduce an overhead that may be considered unacceptable by the users. Moreover, a careless online upgrade may modify the behavior of the system. For instance, under high load MediaWiki disables searching of the database and may even disable write access to the master database. At the egress point, concurrent access to the database by the online system and the data migration process may cause deadlocks. We must therefore avoid imposing an unnecessary overhead through request interception, and we must not lock database rows and tables that may be accessed concurrently by the online system.

Fortunately, the interceptors only have to log incoming requests, and system call interposition has minimal overhead. When using round-robin request tunneling we carefully design the new front-end of interceptor proxies to support the incoming load of the online system. The data migration process uses consistent non-locking reads from the database, which rely on multi-versioning of database tables to query a snapshot of the database that does not reflect the ongoing concurrent transactions. Since we do not set any locks on the tables we access during data migration, we cannot deadlock the database.

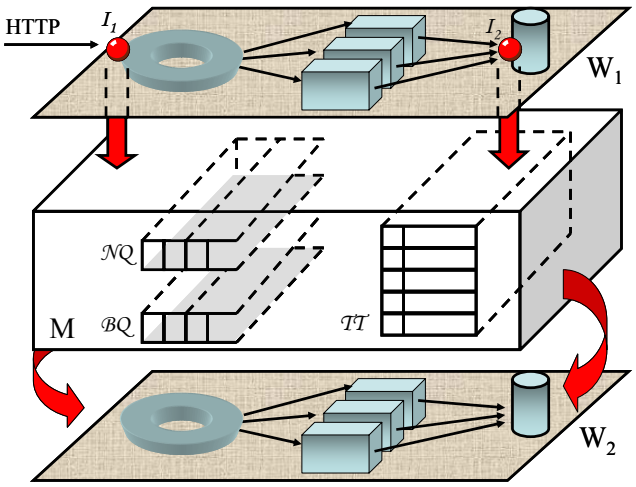

Figure 4. Dependency-agnostic upgrades. The old and new versions are installed and execute in parallel universes W_1 and W_2. MetaMorphMagi intercepts the request flow at the ingress (I_1) and egress (I_2) points of the old version. The rest of W_1 is treated as a black box.

3.1.4 Disabling write access. Before switching over to the new, upgraded system, we must flush all in-progress updates to the persistent data store. In general, this can only be accomplished by disabling write access to the old system (read requests may still go through). This brief period of quiescence prevents any inconsistencies and race conditions during the switchover.

The easiest way to accomplish this is by shutting down the entire old infrastructure. This action would provoke a brief outage of the system, but the outage would be much shorter than in the case when the data transfer is performed while the system is offline. However, such an outage is unnecessary because we do not need to prevent read access to the old system. We could disable write access by using an infrastructure-specific mechanism, such as configuring the master database as read-only. Alternatively, we could instruct the ingress interceptors to reject all the incoming update requests. In this case, these interceptors must use either library interposition or round-robin request tunneling.

3.2 Putting it All Together: a Dependency-Agnostic Upgrade Protocol

We can combine the data migration process described in Section 2 and the techniques presented in Section 3.1 into an online upgrading system for Wikipedia. This online upgrade does not rely on knowledge of the dependencies from within the IT infrastructure,

guarantees data consistency and introduces minimal downtime.

The secret to achieving dependency-agnostic online upgrades is isolating the new version of Wikipedia from the old one by installing the new version in a "parallel universe" – a separate set of machines that cannot communicate directly with the old infrastructure. Figure 4 illustrates this technique. The original Wikipedia runs in an infrastructure called W_1 using the MediaWiki software. The new, TWiki-based encyclopedia runs in a parallel universe W_2. W_1 continues to service incoming requests during the upgrade. The only communication channel between the two universes is via MetaMorphMagi, who migrates the persistent data from W_1 to W_2, monitors the updates handled by W_1 to prevent data-staleness and disables updates to W_1 to enforce quiescence before the switchover.

M^3 uses round-robin request tunneling to intercept the request flow at the ingress points I_1, where the HTTP requests enter the old Wikipedia infrastructure. At the egress points I_2, where persistent data is stored (the master database), we use log-file analysis to monitor the requests. M^3 uses a transfer table *TT* to keep track of the progress of the data migration. When I_2 detects that one of these items has been updated after it was transferred, we invalidate its corresponding entry in *TT* and we (re)schedule it for a fresh transfer W_2. M uses a non-blocking queue *NQ* to monitor in-progress updates. Since M^3 treats the old version of Wikipedia as a black box, with the exception of the ingress and egress points, all the complex dependencies from the infrastructure become 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.

The data migration can proceed as explained in Section 2. The migration driver is part of M^3 and it takes into account the information from I_1 and I_2 when scheduling data items for transfer. The data migration will eventually terminate if the transfer rate exceeds the rate at which previously converted items are invalidated. In the Wikipedia workload, updates (*e.g.* article edits, image uploads) represent only a small fraction of the total number of requests. Moreover, due to the fine granularity of the logical items that we transfer and because the conversion process can be parallelized easily, we are able to satisfy this condition even when the incoming load approaches 30,000 requests/s.

Some requests require special attention. For instance, MediaWiki allows users to change their names. As a user is uniquely identified by the `page_id` field from the MediaWiki database, changing the username is as simple as updating a field in the `user` table. In TWiki there is no easy way to rename a user, as users are uniquely identified by their usernames and topics are signed with the usernames of the people who have created the corresponding revisions. When I_2 determines that a username has changed, we must invalidate all the pages that the user has edited; all the revisions of those pages will be migrated anew and they will be signed with the updated username. This invalidates a lot of data, but fortunately user renaming requests occur infrequently.

We have a similar problem when an article is renamed. MediaWiki creates a redirect page with the old name and a link to the article with the new name. In TWiki, topics are uniquely identified by the topic names, which are also the names of the files where the content is stored. However, this case is easier because the TWiki does not record the topic names anywhere else, so we can create a symbolic link on the filesystem to mirror the behavior of the MediaWiki redirect page.

When the data migration is complete, we can switch from W_1 to W_2. However, we can only perform the switchover when the two universes are in a consistent state. We therefore disable write access to W_1 and we flush all the caches from the infrastructure. The persistent state of W_1 is frozen and the update requests arriving at I_1 are queued inside the middleware (read requests can still go through). When all the outstanding updates have been committed to the database and transferred to W_2, the persistent states of the two universes are synchronized. Note that, while this brief period of quiescence introduces downtime, this does not necessarily mean loss of data: we log the update requests in BQ and we apply them later. After the switchover, all the queued requests as well as all the new requests for URLs from W_1 are converted and redirected to W_2.

Even though the upgrade is performed online, the switchover is not completely transparent to the users. We discard all the volatile state, such as user sessions, and users will be required to log in again. As we cannot migrate hashed data between universes, the users need to reset their passwords before using the new TWiki-based Wikipedia. The switchover is less intrusive for the anonymous users reading Wikipedia articles because they do not have sessions or passwords to reset.

4 Online Is Not Enough: How to Validate an Upgrade and Other Stories

In practice, a major upgrade such as the one described in this paper must be thoroughly tested and validated before the switchover. If the results are deemed unsatisfactory, the administrators must be able to roll

back the upgrade and revert to the old system without any data loss. Our approach for performing online upgrades avoids the problem of dependencies between distributed components by isolating the new version of the infrastructure in a "parallel universe" that does not communicate with the old version. This isolation also allows us to test the new version by injecting faults and running pre-defined traces on the new version without disrupting the functionality of the original system. While performing these tests, we stop the data migration; we can restart it later to capture the effects of the updates processed by the old version while we were running tests on the new one.

It is also possible to execute the new version in parallel with the old one, send the requests intercepted at I_1 to both versions and cross-check the two outputs to validate the upgrade. However, once we start executing in parallel, the states of the two parallel will not be perfectly synchronized anymore because of behavioral differences between the two systems. For example, due to the different access control frameworks of MediaWiki and TWiki, the request to delete a page may succeed in one version and may fail in the other one! Subsequent requests for that page will lead to further state divergence between the versions. This state divergence is acceptable and even desirable because the modified behavior could have been the very reason for initiating the upgrade. When validating the upgrade by running the two versions in parallel, we cannot simply compare the outputs of the two versions. Instead, we must build a model of desirable behavior for the new version of the infrastructure and compare the observed behavior with the properties of this model.

In this paper, we have assumed that the parallel universe where the new version will be installed is a separate physical infrastructure, completely different from the original system and built using new hardware. This scenario is relevant for the situations where the administrators take advantage of the major software upgrade to renew the hardware as well. When the cost of duplicating the hardware is not acceptable, we may consolidate the servers using virtual machines. In this case, we will have a virtual parallel universe, which can provide a similar functional isolation from the old version (*i.e.* we can prevent the old and new versions from communicating with each other by creating two virtual networks). However, this solution may not offer good performance isolation between the two versions if the combined incoming load exceeds the capacity of the infrastructure. When using a virtual parallel universe, we must be careful not to introduce performance dependencies that may affect the behavior of the upgraded system and the data consistency.

5 Related Work

In large-scale enterprise systems, fine-grained changes are performed through rolling upgrades, replacing one component at a time (which means the old and new versions must be able to coexist and interact) [4, 19, 20]. Major upgrades, however, are usually implemented by taking the system off-line during off-peak hours and performing the upgrade and data migration tasks on the inert system (which guarantees downtime) [2, 19, 21]. Process migration across different hosts has been studied extensively for improving availability and providing load-balancing [22]; however, process migration does not support version changes.

Kramer and Magee [23] note that faults, as well as online upgrades, might have a disruptive effect on the functionality of a distributed system, and that the techniques to mitigate these problems could be combined in a unified framework. For instance, a change-management system that totally separates the functional application concerns from the configuration management concerns (such as Kramer and Magee's Conic system), can provide a good basis for implementing fault recovery [23]. Conversely, an infrastructure built for fault-tolerance can provide a good basis for online upgrades because of the inherent redundancy [24, 25]. For example, if the new version of a component or subsystem that is upgraded is fully backwards-compatible with the old version (*i.e.* the semantic, behavioral and interface differences refer only to new functionality, inexistent in the old version), upgrades are a special case of fault recovery: the component is upgraded during a special maintenance window that the system treats as a partial outage [26]. In general, however, it is impossible to guarantee 100% backwards compatibility; experimental studies show that the vast majority of breaking changes are due to refactorings (modifications of the program structure, not intended to change its behavior) [6] or to unintended side-effects that applications rely upon [5].

Dependency-management is a difficult problem even in single-host operating systems. Installing or upgrading an application often disables other applications and services due to shared-library dependencies [5, 27]. Dependencies on configuration settings are even more intimidating because of the fine granularity, the lack of effective dependency-bookkeeping and the sheer quantity of configuration data (comparable to the information stored in the human genome) [28]. For these reasons, best practices in IT system administration recommend the use of a Configuration Management Database (CMDB) that centralizes all the dependency information in the system [29]. Dependencies can be

formally captured using aspects, closures and promises [30]. Unfortunately, complete dependency information cannot be detected automatically through either static analysis [31] or run-time monitoring (which is a best-effort approach) [32]. If this information is specified manually, it might drift from the real state of dependencies, *e.g.* repositories are known to contain metadata inconsistencies that lead to version skew [7].

Existing approaches for online upgrades simplify the problem by requiring semantic dependencies to be specified by the programmer [4], by constraining the communication to typed message-passing channels [33] or by relying on complete dependency reification [34]. Such constraints render these approaches impractical and, which explains why they have not gained a widespread acceptance in the IT industry [35]. The most promising solution, advanced by several researchers [36] and software vendors [37-39], is to run each application in a separate virtual container that prevents communication or cross-couplings between unrelated processes. This is achieved by introducing an indirection layer that provides a unique view of the system resources in each container. These techniques are closest to our protocol for dependency-agnostic upgrades in distributed systems, which was introduced in [8].

Epilogue

Online software upgrades are essential for managing complex IT infrastructures and reducing the administrative costs. In this paper, we examine a hypothetical major upgrade in a realistic IT infrastructure, and we emphasize the significant differences between the two systems that the upgrading process must compensate for. We also present the major building blocks of an online upgrading system called MetaMorphMagi, and we explain how we implement this system by extending an industry-standard, offline data migration process. The resulting online upgrade can tolerate any number of hidden dependencies between components of the infrastructure because the new version is isolated in a "parallel universe". We show that this approach allows us to perform the upgrade with no data loss and minimal downtime. Such an upgrade is not a surgical procedure and is probably unsuitable for regular maintenance activities such as applying security patches. MetaMorphMagi is most useful for large-scale, distributed upgrades because it eliminates the downtime and it reduces the administrative burden by eliminating the need for dependency tracking.

Acknowledgments. We thank Zhengheng Gho and Sreevishnu Byrakur for their help with this project.

References

[1] C. Koch, "AT&T Wireless Self-Destructs," *CIO Magazine*, Apr 2004, http://www.cio.com/archive/041504/wireless.html.
[2] D. Scott, "NSM: Often the Weakest Link in Business Availability," Gartner Group AV-13-9472, July 2001.
[3] D. E. Lowell, Y. Saito, and E. J. Samberg, "Devirtualizable virtual machines enabling general, single-node, online maintenance," *Conference on Architectural Support for Programming Languages and Operating Systems*, pp. 211-223, 2004.
[4] M. Segal and O. Frieder, "On-the-fly program modification: Systems for dynamic updating," *IEEE Software*, vol. 10, pp. 53-65, 1993.
[5] R. Anderson, "The End of DLL Hell," *MSDN Magazine*, 2000.
[6] 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.
[7] J. Hart and J. D'Amelia, "An analysis of RPM validation drift," in *LISA*, Philadelphia, PA, 2002, pp. 155-166.
[8] T. Dumitraş, J. Tan, Z. Gho, and P. Narasimhan, "No More HotDependencies! A Case for Dependency-Agnostic Online Upgrades in Distributed Systems," in *Workshop on Hot Topics in System Dependability*, Edinburgh, Scotland, 2007.
[9] http://meta.wikimedia.org/wiki/Wikimedia_servers.
[10] MediaWiki, http://www.mediawiki.org/wiki/MediaWiki.
[11] TWiki, http://twiki.org/.
[12] Squid Web-Proxy Cache, http://www.squid-cache.org/.
[13] Apache HTTP Server, http://httpd.apache.org/.
[14] Linux Virtual Server, http://www.linuxvirtualserver.org/.
[15] Apache Lucene, http://lucene.apache.org/.
[16] W. F. Tichy, "RCS - A System for Version Control," *Software - Practice and Experience*, vol. 15, pp. 637-654, 1985.
[17] M. Burgess, *Principles of Network and System Administration*: Wiley, 2004.
[18] J. R. Levine, *Linkers and Loaders*. San Francisco, CA: Morgan Kaufmann Publishers, 2000.
[19] E. A. Brewer, "Lessons from Giant-Scale Services," *IEEE Internet Computing*, vol. 5, pp. 46-55, 2001.
[20] S. Ajmani, B. Liskov, and L. Shrira, "Modular Software Upgrades for Distributed Systems," in *European Conference on Object-Oriented Programming*, Nantes, France, 2006.
[21] S. Traugott and J. Huddleston, "Bootstrapping an Infrastructure," in *Large Installation System Administration Conference*, Boston, MA, 1998.
[22] D. Milojicic, F. Douglis, Y. Paindaveine, R. Wheeler, and S. Zhou, "Process Migration Survey," *ACM Computing Surveys*, vol. 32, pp. 241–299, 2000.
[23] J. Kramer, J. Magee, and A. Young, "Towards Unifying Fault and Change Management," in *2nd IEEE Workshop on Future Trends of Distributed Computing Systems in the 1990s*, Cairo, Egypt, 1990, pp. 57-63.
[24] L. E. Moser, P. M. Melliar-Smith, P. Narasimhan, L. A. Tewksbury, and V. Kalogeraki, "Eternal: fault tolerance

and live upgrades for distributed object systems," in *DARPA Information Survivability Conference and Exposition (DISCEX 00)*, Hilton Head, SC, 2000, pp. 184 - 196.

[25] T. Bloom and M. Day, "Reconfiguration in Argus," in *Workshop on Configurable Distributed Systems*. London, England, 1992, pp. 176-187.

[26] T. Limoncelli, "Site Reliability at Google/My First Year at Google," *Invited Talk at Large Installation System Administration Conference*, 2006.

[27] Y. Sun and A. Couch, "Global Impact Analysis of Dynamic Library Dependencies," in *Large Installation System Administration Conference*, Berkeley, CA, 2001.

[28] 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 *LISA*, 2003, pp. 159-172.

[29] Office of Government Commerce, "Information Technology Infrastructure Library (ITIL)," 2001.

[30] M. Burgess and A. Couch, "Modelling Next Generation Configuration Management Tools," in *Large Installation System Administration Conference*, Washington, DC, 2006, pp. 131-147.

[31] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson, "Automated Detection of Refactorings in Evolving Components," in *ECOOP*, Nantes, France, 2006, pp. 404-428.

[32] 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 *ICAC*, 2004, pp. 106-113.

[33] J. Kramer and J. Magee, "The Evolving Philosophers Problem: Dynamic Change Management," *IEEE Transactions on Software Engineering*, vol. 16, pp. 1293-1306, 1990.

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

[35] M. E. Segal, "Online software upgrading: new research directions and practical considerations," *Computer Software and Applications Conference*, pp. 977-981, 2002.

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

[37] N. Ruest, "Software Virtualization - Ending DLL Hell Forever," in *Microsoft Management Summit*, San Diego, CA, 2006.

[38] Microsoft SoftGrid, http://www.softricity.com/.

[39] Altiris Software Virtualization Solution, http://www.altiris.com/Products/SoftwareVirtualizationSolution.aspx.

Appendix: Data Mapping from MediaWiki to TWiki

Item in MediaWiki	Backlink	Mapped Item in Twiki	Mapping Match (Good, Moderate, Poor)	Comments
page_id				
page_namespace		<Web>	Moderate	
page_title		<Web>.<TopicName>	Good	
page_restrictions		<Web>.<TopicName>.Permissions	Poor	
page_counter		<WebStatistics>.entries	Good	
page_is_redirect		NO EQUIVALENT		
page_is_new		NO EQUIVALENT		
page_random		NO EQUIVALENT		
page_touched		NO EQUIVALENT		
page_latest		INTERNAL STATE		
page_len		INTERNAL STATE		
page_no_title_convert		INTERNAL STATE		
rev_id		INTERNAL STATE		
rev_page	page_id	<Web>.<TopicName>.Revisions	Good	
rev_text_id	old_id	<Web>.<TopicName>.Revisions	Good	
rev_comment		NO EQUIVALENT		
rev_user	user_id	<Web>.<TopicName>.TOPICINFO	Good	
rev_user_text		<Web>.<TopicName>.TOPICINFO	Moderate	
rev_timestamp		<Web>.<TopicName>.TOPICINFO	Good	
rev_minor_edit		<Web>.<TopicName>.TOPICINFO	Moderate	RepRev
rev_deleted		NO EQUIVALENT		Not sure
old_id		INTERNAL STATE		
old_text		<Web>.<TopicName>.Revisions	Good	
old_flags		<Web>.<TopicName>.Revisions	Moderate	Not sure of specifics
ar_namespace		<Web>	Moderate	Special - in <TrashWebName>
ar_title		<Web>.<TopicName>	Good	
ar_text	old_id	<Web>.<TopicName>.Revisions	Good	
ar_comment		NO EQUIVALENT		
ar_user	user_id	<Web>.<TopicName>.TOPICINFO	Good	
ar_user_text		<Web>.<TopicName>.TOPICINFO	Moderate	
ar_timestamp		<Web>.<TopicName>.TOPICINFO	Good	
ar_minor_edit		<Web>.<TopicName>.TOPICINFO	Moderate	RepRev
ar_flags		<Web>.<TopicName>.Revisions	Moderate	Not sure of specifics
ar_rev_id		UNDOCUMENTED		
ar_text_id		UNDOCUMENTED		
rc_id		INTERNAL STATE		
rc_timestamp		<Web>.<TopicName>.TOPICINFO	Good	
rc_cur_time		DEPRECATED		
rc_user	user_id	<Web>.<TopicName>.TOPICINFO	Good	
rc_user_text		<Web>.<TopicName>.TOPICINFO	Moderate	
rc_namespace		<Web>	Moderate	
rc_title		<Web>.<TopicName>	Good	
rc_comment		NO EQUIVALENT		
rc_minor		<Web>.<TopicName>.TOPICINFO	Moderate	RepRev
rc_bot		NO EQUIVALENT		
rc_new		NO EQUIVALENT		
rc_cur_id	page_id	<Web>.<TopicName>	Good	
rc_this_oldid	old_id	<Web>.<TopicName>.Revisions	Good	Current text
rc_last_oldid	old_id	<Web>.<TopicName>.Revisions	Good	Previous text
rc_type		<Web>.<Statistics>.entries	Moderate	
rc_moved_to_ns		DEPRECATED		
rc_moved_to_title		DEPRECATED		

rc_patrolled		NO EQUIVALENT		
rc_ip		NO EQUIVALENT		
rc_old_len		INTERNAL STATE		
rc_new_len		INTERNAL STATE		
user_id		INTERNAL STATE		
user_name		SystemWeb.<UserName>, SystemWeb.UserTopic	Good	
user_real_name		SystemWeb.<UserName>, SystemWeb.UserTopic	Good	
user_password		HASHED		
user_newpassword		HASHED		
user_email		SystemWeb.UserTopic	Good	
user_options		UNDOCUMENTED		
user_touched		TRANSIENT		
user_token		TRANSIENT		
user_email_authenticated		TRANSIENT		
user_email_token		TRANSIENT		
user_email_token_expires		TRANSIENT		
user_registration		SystemWeb.UserTopic	Good	Date registered
user_newpass_time		TRANSIENT		
user_editcount		NO EQUIVALENT		
ug_user	page_id	SystemWeb.<NameOfGroup>	Moderate	
ug_group		SystemWeb.<NameOfGroup>	Moderate	
ss_row_id		NO EQUIVALENT		
ss_total_views				
ss_total_edits				
ss_good_articles				
ss_total_pages				
ss_users				
ss_admins				
ss_images				
hc_id		<Web>.<Statistics>.entries	Good	
log_type		<Web>.<Statistics>.entries	Poor	
log_action		<Web>.<Statistics>.entries	Poor	
log_timestamp		<Web>.<Statistics>.entries	Good	
log_user	user_id	<Web>.<Statistics>.entries	Good	
log_namespace	page_namespace	<Web>.<Statistics>.entries	Moderate	
log_title	page_title	<Web>.<Statistics>.entries	Good	
log_comment		NO EQUIVALENT		
log_params		UNDOCUMENTED		
log_id		INTERNAL STATE		
wl_user		<Web>.<Notification>	Good	
wl_namespace		<Web>.<Notification>	Good	
wl_title		<Web>.<Notification>	Good	
wl_notificationtimestamp		<Web>.<Notification>	Poor	