

# Staged Deployment in Mirage, an Integrated Software Upgrade Testing and Distribution System\*

Olivier Crameri  
EPFL, Lausanne, Switzerland  
olivier.crameri@epfl.ch

Nikola Knežević  
EPFL, Lausanne, Switzerland  
nikola.knezevic@epfl.ch

Dejan Kostić  
EPFL, Lausanne, Switzerland  
dejan.kostic@epfl.ch

Ricardo Bianchini  
Rutgers University, NJ, USA  
ricardob@cs.rutgers.edu

Willy Zwaenepoel  
EPFL, Lausanne, Switzerland  
willy.zwaenepoel@epfl.ch

## ABSTRACT

Despite major advances in the engineering of maintainable and robust software over the years, upgrading software remains a primitive and error-prone activity. In this paper, we argue that several problems with upgrading software are caused by a poor integration between upgrade deployment, user-machine testing, and problem reporting. To support this argument, we present a characterization of software upgrades resulting from a survey we conducted of 50 system administrators. Motivated by the survey results, we present Mirage, a distributed framework for integrating upgrade deployment, user-machine testing, and problem reporting into the overall upgrade development process. Our evaluation focuses on the most novel aspect of Mirage, namely its staged upgrade deployment based on the clustering of user machines according to their environments and configurations. Our results suggest that Mirage's staged deployment is effective for real upgrade problems.

## Categories and Subject Descriptors

C.2.4 [Computer-Communication Networks]: Distributed Systems; D.4.5 [Operating Systems]: Reliability; D.4.5 [Operating Systems]: Systems Programs and Utilities

## General Terms

Management, Reliability

## Keywords

Upgrade testing, Clustering of machines, Staged software upgrade deployment

---

\*This research is supported in part by the Hasler Foundation (grant 2103).

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

SOSP'07, October 14–17, 2007, Stevenson, Washington, USA.  
Copyright 2007 ACM 978-1-59593-591-5/07/0010 ...\$5.00.

## 1. INTRODUCTION

Software upgrades involve acquiring a new version of an application, an improved software module or component, or simply a “patch” to fix a bug, and integrating it into the local system. Upgrades for Linux and Windows are released almost every month, while upgrades of user-level utilities and applications are frequent as well.

For software users, system administrators, and operators (hereafter collectively called users, for conciseness), integrating upgrades is often an involved proposition. First, an upgrade can affect multiple applications, and all of them may need to be tested for correct behavior after the upgrade. Second, the effects of buggy upgrades may need to be rolled back. Third, upgrades are prone to a variety of incompatibility and unexpected behavior problems, especially when components or modules, rather than entire applications or systems, are upgraded.

For software vendors, distributors, and open-source contributors (hereafter collectively called vendors, for conciseness), it is difficult to deploy the upgrades with high confidence that they will integrate properly into the users' systems and that they will behave as users expect. The vendors simply cannot anticipate and test their upgrades for all the applications and configurations that may be affected by the upgrades at the users' machines. To lessen this problem, vendors sometimes rely on “beta testing”. However, beta testers seldom provide complete coverage of the environments and uses to which upgrades will be exposed. Another approach to reduce integration problems is for vendors to use package-management systems to deploy their upgrades. Dependency enforcement in these systems, however, only tries to enforce that the right packages are in place. They do not provide any help with locally testing the upgrades for correct behavior or reporting problems back to vendors.

When integration problems occur, vendors receive only limited and often unstructured information to pinpoint and correct the problems. Most of the time, the vendor at best receives core dumps of applications that crash because of the upgrade or often incomplete problem reports posted to mailing lists or Web forums. Because of this limited information, it may take several iterations of deployment, testing, and debugging before an upgrade becomes useful to all of its intended users. Furthermore, because reports may come from any source, e.g., multiple beta testers or regular users with similar installations, there is often much repetition in

the information received by the vendor, requiring significant human resources to filter out.

Although much anecdotal evidence suggests a high frequency of upgrade problems, there is surprisingly little information in the literature characterizing upgrades in detail. To start bridging this gap in the literature, we perform a characterization of upgrades, using responses to a survey that we conduct among 50 system administrators. The results confirm that upgrades are done frequently, that problems are quite common, that these problems can cause severe disruption and that therefore upgrades are often delayed, and that users seldom fully report the problems to the vendor. Additionally, and of particular importance to this paper, many of the problems are caused by differences between the environment at the vendor and at the users. Broken dependencies, incompatibilities with legacy applications, and improper packaging issues are among this class of problems. While other upgrade problems, such as bugs or removed behaviors, can presumably be addressed by better development and testing methods at the vendor, the problems related to vendor-user environment differences necessitate a novel approach that takes into account these differences.

Mirage is a framework that integrates upgrade deployment, user-machine testing, and problem reporting. In the current state of the art, these three activities are loosely linked in an ad-hoc manner. Mirage integrates them into a structured and efficient upgrade development cycle that encompasses both vendors and users. Mirage deploys complex upgrades in stages based on the clustering of the user machines according to their environments, tests the upgrades at the users' machines, and sends information back to vendors regarding the success/failure of each user's upgrade, including, in case of failure, the context in which the upgrade fails.

A key goal of Mirage is to guarantee that upgrades behave properly before they are widely deployed. Clustering machines as in Mirage helps achieve this goal, while simplifying debugging at the vendor. We also identify a fundamental tradeoff between the number of upgrade problems at the user machines and the speed of deployment. To address this tradeoff, Mirage provides a few abstractions on top of which different staged deployment protocols can be implemented.

We evaluate Mirage's clustering algorithm with respect to real upgrades and problems reported in our survey. The results demonstrate that our algorithm can cluster the users' machines effectively. Our algorithm works best when used in combination with vendor-provided information about the importance of certain aspects of the application's environment. Further, our simulations show that deploying upgrades in stages significantly reduces the amount of testing effort required of the users, while quickly deploying the upgrade at a large fraction of machines. Although we occasionally delay deploying certain upgrades, by increasing the users' confidence, they may actually apply the upgrades sooner.

The rest of this paper is organized as follows. The next section characterizes software upgrades in detail based on our survey. Section 3 presents Mirage and its key concepts. Section 4 evaluates our clustering algorithm in the context of realistic upgrade problems, and analyzes the performance of two staged deployment protocols using a simulator. Section 5 overviews the related work. Finally, Section 6 concludes the paper.

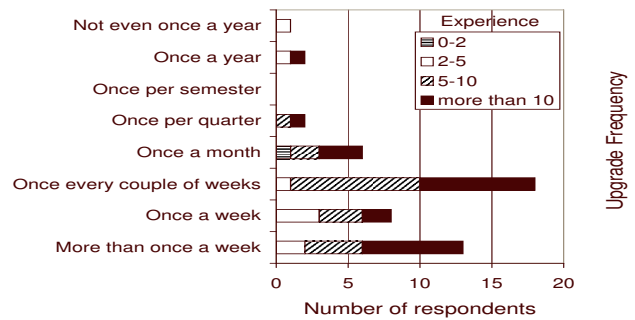


Figure 1: Upgrade frequencies.

## 2. CHARACTERIZING UPGRADES

### 2.1 Methodology

We conducted an online survey to estimate the frequency of software upgrades, the reasons for the upgrades, the frequency of software upgrade problems, the classes of problems that occur, and the frequency of the different classes. We posted the survey (available at [33]) to the USENIX SAGE mailing list, the Swiss network operators group, and the EliteSecurity forum in Serbia; we received 50 responses.

The majority of the respondents to our survey (82%) have more than five years of administration experience, and 78% manage more than 20 machines. Our respondents were allowed to specify multiple, non-disjoint choices of operating systems, making it difficult to correlate upgrade problems with an operating system version. In our survey, 48 of the administrators manage a Linux system or some other UNIX-like system, 29 administrators manage Windows desktops and server platforms, while 12 respondents manage Mac OS machines.

### 2.2 Survey Results

**Frequency of software upgrades.** As Figure 1 demonstrates, upgrades are frequent. Specifically, 90% of administrators perform upgrades once a month or more often.

**Reasons for upgrades.** We asked respondents to rank (from 1 to 5, 1 being most important) each of five possible reasons: 1) bug fix, 2) security patch, 3) new feature, 4) user request, and 5) other reason. Security fixes get the highest priority (average rank 1.6), followed by bug fixes (average rank 2.2) with user requests and new features trailing (average ranks of 3.3 and 3.5, respectively). Given that the primary reason for performing an upgrade is a security fix, it can be harmful if administrators delay upgrading.

**Upgrade installation delays.** Nevertheless, 70% of our respondents report that they refrain from installing a software upgrade, regardless of their experience level. This is the case even though 70% of administrators have an upgrade testing strategy (Figure 2 shows these results). An overwhelming majority (86%) of administrators use the software packaged with the operating system to install upgrades.

**Upgrade failure frequency.** We asked respondents to estimate this rate. As Figure 3 shows, 66% of them estimate that between 5 and 10% of the upgrades that are applied have problems. Given the potential damage and extra effort that a faulty upgrade can entail (discussed next), such a failure rate is unacceptable. Over all the responses, the average failure rate is 8.6% and the median is 5%.

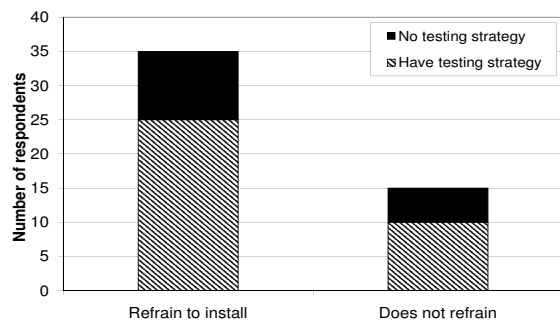


Figure 2: Reluctance to upgrade.

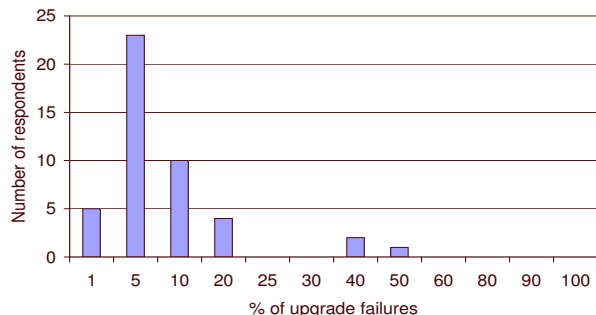


Figure 3: Perceived upgrade failure rate.

Another reason for avoiding upgrades may be the damage that a faulty upgrade can cause. An important fraction of the administrators (48%) experience problems that pass initial testing, with 18% reporting catastrophic upgrade failures.

**Testing strategies.** The majority of the testing strategies our respondents use involve having a testing environment (25 respondents). Six of the administrators report testing the upgrade on a few machines, then moving on to a larger fraction, before finally applying the upgrade to the entire machine set. However, only 4 respondents use an identical machine configuration in the testing environment. Two of the respondents rely on reports of successful upgrades on the Internet without an additional testing strategy. Although 70% of the respondents have an upgrade testing strategy, only 18% of the administrators report that they do not have upgrade problems because they use it.

**Causes of failed upgrades.** We asked the administrators to assign a rank to each of the failure categories that we provided, including an optional “other categories” field. Our respondents identified broken dependencies, removed behavior, and buggy upgrades as the most prevalent causes (average ranks of 2.5, 2.5 and 2.6, respectively). They deemed incompatibility with legacy configurations and improper packaging to be less common (average ranks of 3.1 and 3.2, respectively). We detail some of the reported problems in Section 2.3. Here we conclude that no single cause of upgrade failure is dominant.

**Problem reporting.** Only half of the respondents consistently report upgrade problems to the vendor. When they do so, the information they typically include is the versions of the operating system and related software, type of hardware, and any error messages.

## 2.3 Categories of Upgrade Problems

Here we describe the major categories of upgrade problems (in decreasing order of importance assigned by respondents) and a few examples. We combined the reported problems with some that we found on Web forums.

**Broken dependency.** This category captures subtle dependency issues that often cause severe disruption. Typically, application  $A_X$  is compiled with a specific library  $L_Y$  or depends on some application  $A_Y$ . Upgrading some other application  $A_Z$  that depends on  $L_Y$  (or  $A_Y$ ) can upgrade  $L_Y$  (or  $A_Y$ ) as well, causing application  $A_X$  to fail. For example, when upgrading the MySQL DBMS from version 4 to 5, the PHP scripting language crashes if it was compiled with MySQL support [24]. PHP still tries to use libraries from the old version. Some of these broken dependency issues could be caught by the package management system. However, package-management systems cannot catch these issues if the administrator upgrades some files manually.

**Important feature was removed, or behavior was altered.** A significant number of upgrade problems is caused by the removal of an important feature of the software. A related problem is altered behavior of an application or a changed API. An example of this category is an upgrade of PHP4 to PHP5 that caused some scripts (.php pages) to stop working due to a new Object Model [28]. The problems in this category cannot be alleviated by improved packaging, as it becomes almost impossible to search and upgrade all cases of deprecated behavior across all user files.

**Buggy upgrades.** A natural category of the failed upgrades captures the bugs in the upgrade itself. For example, Firefox crashes when displaying some Web pages with JavaScript on Ubuntu, after the 1.5.0.9 upgrade is applied [12]. Some of the problematic upgrades our survey respondents reported have serious consequences, such as wireless, RAID, and other driver upgrades that caused the system to fail to boot.

**Incompatibility with legacy configurations.** Another category of upgrades involves machine-specific data that causes an upgraded application to fail. This kind of data includes environment variables and configuration files. For example, the upgrade of Apache from 1.3.24 to 1.3.26 caused Apache configurations that included an access control list to fail [3]. The contents of the included file had to be moved to the main configuration file for Apache to start working again. Again it is difficult for upgrade packaging to handle all possible configurations at user machines.

**Improper packaging.** This category of upgrade problems occurs when the creator of the new package (that contains the upgrade) does not correctly upgrade all application components. Typically, this includes an omission of some application file, as it was the case with SlimServer 6.5.1. Here, the database was not automatically upgraded during the upgrade process, and the server consequently would not start due to a changed database format. Issues like this one can be addressed by a more careful packaging procedure.

**Minor problems.** The final category comprises problems that are not obvious bugs but represent an annoyance. Often, the problem arises when a cached file is overwritten instead of upgraded. Some of these seemingly minor problems can have serious consequences. For example, one administrator was unable to log into the Zertificon application due to an administrator password that was overwritten.

## 2.4 Discussion

We did not seek to perform a comprehensive, statistically rigorous survey of upgrade management in the field. Our main goals were to motivate Mirage, while collecting a sampling of data on real upgrades and their problems. These restricted goals allowed to focus on a group of administrators, mostly from USENIX SAGE, that volunteered information about their practical experiences. Admittedly, this approach may have been affected by self-selection, bias, and coverage problems. Nevertheless, our survey does provide plenty of information on real upgrades and is broader than the few comparable works in the literature [4, 30], which focused solely on security upgrades and have their own limitations. In fact, these other works confirm some of our observations: Beattie et al. [4] state that the initial security upgrade failure rate is even higher than that reported by our respondents for all upgrades. An inspection of 350,000 machines by Secunia [30] finds 28% of all major applications to be lacking the latest security upgrades, which is in line with the high fraction of our respondents who delay upgrades.

## 3. MIRAGE

### 3.1 Overview

Many approaches are available to reduce the burden of upgrades. For instance, improved development and testing methods at the vendor may reduce the number of buggy upgrades. As can be seen, however, from the results of the survey, several of the frequently occurring categories of upgrade problems are due to differences between the development and testing environment at the vendor and the many different environments at user machines. Broken dependencies and incompatibility with legacy configurations clearly fall into this category, whereas improper packaging may also result from environmental differences.

The goal of the Mirage project is to reduce the upgrade problems associated with such environmental differences between vendors and users. To do so, the Mirage framework provides three components that cooperate in a structured manner: staged deployment, user-machine testing, and problem reporting (see Figure 4).

To correct problems with environmental differences between vendor and users, an upgrade would need to be tested in all possible user environments. In-house vendor testing of all possible user environments is infeasible, because the vendor cannot possibly anticipate all possible uses and environments. Indiscriminate testing at all user sites is also undesirable, because the problematic upgrades may inconvenience a large number of users. The current practice of beta-testers tries to reduce this inconvenience, but provides only limited coverage.

To address these issues, Mirage provides staged deployment. Machines are clustered based on their environments such that machines in the same cluster are likely to behave the same with respect to an upgrade. Within clusters, one or a few machines test the upgrades first, before other machines are allowed to download and test the upgrade. Finally, staged deployment allows control over the order in which upgrades are deployed to clusters.

With staged deployment, the vendor does not need to anticipate all possible user environments, but at the same time indiscriminate user-machine testing is avoided. With clustering, the representatives can provide better coverage than

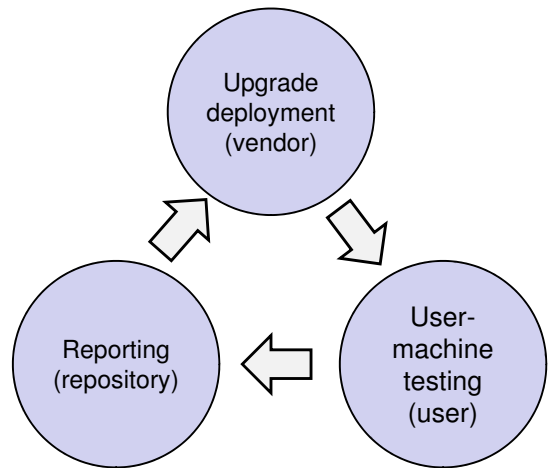


Figure 4: High-level overview of Mirage.

current beta-testing. As the survey shows, users are typically willing to wait to install upgrades, so the extra delay potentially introduced by staging is acceptable, especially in light of the reduction in inconvenience.

In addition to staged deployment, Mirage provides a user-machine testing facility that tries, as much as possible, to automate the upgrade testing in the local environment and to improve its coverage. Such testing may, however, never be fully automated or completely conclusive, but we do not rely on that.

Finally, to improve over the ad-hoc reporting seen in the survey, Mirage provides a structured reporting facility, allowing users to deposit the results of their testing in a repository accessible to the vendor.

Our current Mirage implementation is geared towards Linux, but the framework is also applicable to Windows.

### 3.2 Deployment Subsystem

Mirage provides a small number of abstractions to support staged deployment. Upon these basic abstractions, vendors can build different deployment protocols to reach different objectives. Objectives may include reducing upgrade overhead (which we define as the number of machines that test a faulty upgrade), reducing upgrade latency (the delay between the upgrade being available at the vendor and it being applied at the user), reducing the number of redundant problem reports, front-loading the problem debugging effort, or combinations thereof.

#### 3.2.1 Abstractions

The three basic deployment abstractions provided by Mirage are: clusters of deployment, representatives, and distance between vendor and clusters. (1) User machines are clustered into groups that are likely to behave similarly with respect to upgrades of a particular application. (2) Each cluster has at least one representative machine. The representative tests the upgrade before any of the non-representative machines in its cluster do, playing a role similar to today's beta-testers. (3) In addition, a measure of distance may be defined between the vendor and a cluster, indicating the degree of difference between the vendor's environment and that of the machines in the cluster. Intuitively, if a machine is more dissimilar from the vendor's machine,



the likelihood of a problem with the upgrade is higher. The vendor can take this additional information into account in guiding the deployment protocol. These three abstractions, and in particular the underlying clustering algorithm, are the main focus of this paper.

Clustering is done per application, but has to be evaluated in the context of a particular upgrade. The quality of the clustering is critical for many of the potential vendor objectives. For example, the upgrade overhead is directly related to the quality of the clustering. To see this relationship, consider the following types of clustering assuming that user-machine testing detects an upgrade problem if and only if there is one.

In an *ideal* clustering, all machines in a cluster behave identically with respect to an upgrade and differently from machines in other clusters. If the upgrade behaves correctly at one machine in a cluster, it behaves correctly at all machines in that cluster. Further, all machines at which the upgrade behave correctly are in the same cluster. If the upgrade exhibits a problem, it exhibits that same problem everywhere in the cluster. Moreover, all machines that exhibit that problem are in the same cluster. If other machines exhibit a different problem, they are in a different cluster. Thus, upgrade overhead is limited to the number of representatives, which is minimal, as is the number of reports sent to the upgrade result repository.

While ideal, this form of clustering is difficult to achieve in practice. For this reason, we consider the following slightly less ambitious goal. In a *sound* clustering, all machines in a cluster behave identically with respect to an upgrade, as with ideal clustering, but there may be multiple clusters with the same behavior. In other words, multiple clusters may exhibit correct behavior or the same incorrect behavior. The upgrade overhead is no longer minimal, but as long as the number of clusters with problems remains small relative to the total number of machines, there is considerable improvement.

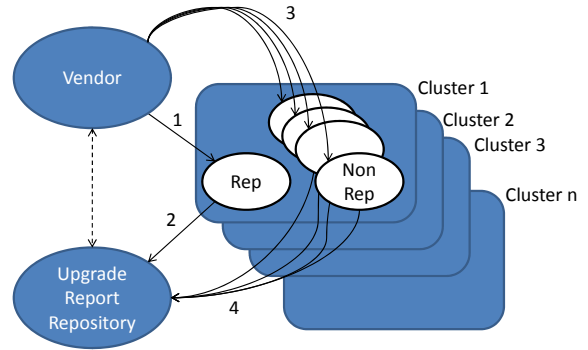
Clustering becomes considerably worse, if machines within the same cluster behave differently with respect to an upgrade. In this *imperfect* clustering, if an upgrade problem does not manifest itself at the representative, non-representatives are bothered with testing incorrect upgrades. This problem can be marginally improved by having a few rather than one representative. If the upgrade fails at the representative but works correctly at some other machines, then upgrades to those machines are needlessly delayed.

### 3.2.2 Protocols

Using these basic abstractions and a particular clustering algorithm, the vendor may implement different deployment protocols to optimize different criteria. In terms of structure, a protocol is defined by the degree of deployment parallelism it embodies and, when certain phases are sequential, the particular sequential orders it imposes.

In any staged deployment protocol, the upgrade is tested on the representative(s) of a cluster before it goes to non-representatives of that cluster. That constraint leaves open many variations in terms of parallelism and ordering.

A vendor may wait for all representatives of all clusters to successfully test an upgrade before it sends it to any non-representatives. Within this basic paradigm, it can send the upgrade in parallel to all or some representatives or sequentially one at the time to each of them. The same applies



**Figure 5: Mirage deployment: clusters of machines, cluster representatives and non-representatives, and cluster ordering. Some interactions of a deployment protocol are also shown.**

applies to the non-representatives: the upgrade may be sent to all or some clusters in parallel, or sequentially to one at a time.

An alternative approach is for the vendor to send the upgrade to the non-representatives of a cluster as soon as testing finishes successfully on the representatives of that cluster. This approach again leaves open the choice of sending the upgrades to the representatives of a cluster sequentially or in parallel.

When deployment should proceed one cluster after the other, the protocol needs to define the exact ordering of cluster deployments. To support this ordering, the vendor can use Mirage’s distance metric. Going from smaller to larger distances may reduce the average upgrade latency, whereas going in the inverse order may allow the vendor to front-load its problem debugging effort.

Different upgrades may be treated differently. In fact, the vendor’s objective for each upgrade typically depends on the characteristics of the upgrade. If the upgrade is a major new release, the vendor may decide to go slowly. If the upgrade is urgent and the vendor has high confidence in its correctness, it may bypass the entire cluster infrastructure, and distribute the upgrade to all users at once. Figure 5 depicts the key Mirage deployment abstractions and a few interactions of a generic deployment protocol.

Mirage does not advocate any specific protocol. In Section 4.3 we evaluate two specific protocols to illustrate some of the possibilities, but many other variations exist.

### 3.2.3 Clustering Machines

For upgrade deployment, Mirage seeks to cluster together user machines that are likely to behave similarly with respect to an upgrade. Since many of the most common upgrade problems result from differences in environment at the vendor and at the user machine, Mirage clusters machines with similar environments. Thus, before clustering, Mirage needs to identify the *environmental resources* on which the application to be upgraded depends. These resources typically include the operating system, various runtime libraries, executables, environment variables, and configuration files that each application uses (the environmental resources on a Windows-based system would include the registry as well).

Next, we detail the steps taken to collect and fingerprint (derive a compact representation) the set of environmental resources. After that, we describe our clustering algorithm.

**Identifying environmental resources.** We instrument, both at the vendor and at the user machines, the process creation, read, write, file descriptor-related and socket-related system calls. For environment variables, we intercept calls to the `getenv()` function in `libc`. Using this instrumentation, we create a log of all external resources accessed. This log may include data files in addition to environmental resources. Clustering becomes ineffective if data files are considered as environmental resources.

We use a four-part heuristic to identify environmental resources among the files accessed by the application in a collection of traces, combined with an API that allows the vendor to explicitly include or exclude sets of files. The heuristic identifies as environmental resources: (1) all files accessed in the longest common prefix of the sequence of files accessed in the traces, (2) all files accessed read-only in all execution traces, (3) all files of certain types (such as libraries) accessed in any single trace, and (4) all files named in the package of the application to be upgraded. In our current implementation, we concentrate on files and environment variables. We plan to address machine hardware properties, network protocols, and other aspects of the environment in our future work.

The first part of the heuristic is based on the observation that the execution of applications typically starts with a single-threaded initialization phase in which the application loads libraries, reads configuration files and environment variables, etc. Most often, no data files are read in this initialization phase, and hence all files accessed in this phase are considered environmental resources. To find the end of the initialization phase, we compute the longest common sequence of files from the beginning that are accessed in every trace.

The second and third parts of the heuristic deal with applications that access environmental resources after the initialization phase, for example, applications that use late binding or load extensions at runtime. To identify those environmental resources, our heuristic considers all files opened read-only after the initialization sequence. To separate environmental resources from read-only data files, we only classify as environmental resources those files that are opened in every execution or those files that are of a few vendor-specified types, such as libraries.

The final part of our heuristic includes all files bundled in the package of the application to be upgraded.

The heuristic may erroneously designate certain files as environmental resources. For instance, the initialization phase may access a log or some initial data files, or sample data files may also be included in a package. Conversely, certain environmental resources are accessed in read-write mode and therefore not included by the heuristics, typically because the files contain both (mutable) data and configuration information. To address these issues, a simple API provided by Mirage allows the vendor to include or exclude files or directories. By default, we exclude some system-wide directories, such as `/tmp` and `/var`.

As we show in Section 4.1, our heuristic combined with a very small number of vendor-provided API directives allows Mirage to correctly identify the environmental resources of popular applications with no false positives and no false negatives.

**Resource fingerprinting.** To produce inputs to the clustering algorithm, we need a concise representation (a finger-

print) for each environmental resource at the vendor and at the user machines. Depending on its type, we have three different ways of fingerprinting a resource. First, for common types such as libraries, Mirage provides a parser that produces the fingerprint. Second, the vendor may provide parsers for certain application-specific resources, like configuration files. Third, if neither Mirage nor the vendor provides a parser for a resource, the fingerprint is a sequence of hashes of chunks of the file that are content-delineated using Rabin fingerprinting. In general, we expect that for many applications the vast majority of the resources are handled by parsers, and we resort to Rabin fingerprinting only in rare circumstances.

A resource’s fingerprint contains a hierarchical set of keys and their values (items for short). The items produced by the parsers have a common structure but a meaning that is dependent on the resource type. Each parser is responsible for determining the granularity of the items that are produced. Indeed, producing fine-grained items might be useful for some types of resources (such as configuration files) and useless for others (such as executables). In addition, some resources may contain user-specific data (such as user names and IP addresses) and other information (such as comments) that are irrelevant in clustering machines; it is up to each parser to extract the useful information from the resource and discard the rest.

The parsers for the most common resource types produce items in the following formats:

- Executables: ExecutableName.FILE\_HASH
- Shared libraries: LibraryName.Version#.HASH
- Text files: Filename.Line#.LINE\_HASH
- Config files: Filename.SectionName.KEY.HASH
- Binary files: Filename.CHUNK\_HASH

In contrast, the content-based fingerprinting creates items of the form: Filename.CHUNK\_HASH. Clearly, these fingerprints are more coarsely grained than what is possible with parsers. In fact, using this method, all the irrelevant data contained in the environmental resources are fingerprinted together with the relevant information. We leverage the publicly available Rabin fingerprint implementation [23], and use the default 4 KB average chunk size.

Once the vendor produces its list of items for the environmental resources on the reference machine, it sends it to the user machines. Each user machine compares the list of items from the vendor to its own list and produces a new list with the set of items that differ from those of the vendor. The new list contains both items present on the reference machine but not on the user machine and vice-versa. This new list is sent back to the vendor, triggering the clustering algorithm.

**Clustering algorithm.** The algorithm is divided into two phases. In the first phase, clustering is performed with respect to the resources for which there are parsers. Because for these resources we have precise information about the relevant aspects of the environment, two machines are assigned to the same cluster if and only if their sets of items that differ from the vendor are the same. We refer to the clusters produced in this first phase as “original clusters”.

In the second phase, the environmental resources for which the vendor does not provide a parser are taken into account. In this phase, we need to decide whether to split the original

clusters based on these resources. Since content-based fingerprints do not leverage semantic information about the resources, they are imprecise representations of the resources' contents. To cope with this imprecision, we use a diameter-based algorithm that is a variation of the QT (Quality Threshold) clustering algorithm [15] (we dismissed the traditional k-means algorithm because it is non-deterministic) on each of the original clusters.

Considering the user machines of only one original cluster at a time, the diameter-based algorithm starts with one machine per cluster. It then iteratively adds machines to clusters while trying to achieve the smallest average inter-machine distance, without exceeding the vendor-defined cluster diameter  $d$ . The distance metric is the Manhattan distance between two machines, defined as the number of different items associated with the resources for which there are no parsers. When the algorithm cannot merge any additional clusters, it moves on to the next original cluster. After the final set of clusters is produced, we also explicitly split clusters that contain machines with different sets of applications with overlapping environmental resources. The final clusters are labeled with their set of differing items.

**Discussion.** At this point, it should be clear that it is advantageous for the vendor to provide parsers for as many environmental resources as possible. Doing so allows the vendor to cluster machines more accurately. In addition, using parsers, the vendor can exercise greater control over the clustering process. For example, the vendor can specify which items it believes to be less important. The vendor can create bigger clusters by removing those items from the set of differing items of each machine. It is also possible to discard only a suffix of some of the hierarchical items. For example, assume that many machines have version 2.4 of libc, but a significant portion of them have a different library hash. This discrepancy probably means that there are machines that have version 2.4 of libc compiled with different flags. In deploying a non-critical upgrade to the graphical interface of Firefox, the vendor might decide to discard this information, thus putting all machines using version 2.4 of libc in one bigger cluster.

Relative to performance and scalability, the user-side fingerprinting and comparison processes are efficient and distributed, regardless of how many parsers are provided. However, producing as many parsers as possible has advantages in terms of the vendor's computational effort. The first phase of the clustering algorithm runs efficiently in time that is proportional to the number of user machines. In contrast, the second phase performs a number of comparisons that is quadratic in the number of machines in each original cluster, and each comparison takes time proportional to the number of items associated with resources for which there are no parsers. Thus, the second phase can become computationally intensive if there are many such resources. The running time of the QT algorithm is also quadratic in the number of machines in each original cluster. In our future work, we plan to develop an efficient incremental reclustering approach, since a relevant change in a machine's environment can change that machine's cluster.

### 3.3 User-machine Testing Subsystem

Our infrastructure for upgrade testing traces the behavior of each application before an upgrade, and compares it to the behavior after the upgrade. The upgrade is performed

in an isolated environment to facilitate rollback in case of problems. Our current approach relies on a problem-free upgrade not changing the I/O behavior of the application. Many upgrades, for instance, those that fix bugs or eliminate security vulnerabilities fall into this category. We discuss upgrades that do change I/O behavior in Section 3.5.

Our testing infrastructure has three interacting subsystems: dependence, trace-collection, and upgrade-validation. The dependence subsystem provides triggers for trace collection, as well as application dependence information that drives upgrade validation. The trace collection subsystem feeds the upgrade-validation subsystem with input and output information. The upgrade-validation subsystem tests each upgrade for correctness.

Next, we detail each of the subsystems in turn.

**Dependence subsystem.** When an upgrade is received, Mirage needs to determine the applications that can be affected by it. To do so, Mirage relies on the dependency information also used by our clustering algorithm (Section 3.2.3).

**Tracing subsystem.** Our trace-collection subsystem intercepts system calls to record the data used as inputs and outputs. The contents of files read need not be recorded, since, as discussed below, the validation subsystem re-executes the read from the same file system state. To collect the inputs and outputs, trace collection extends the same calls instrumented by the dependence subsystem, but also saves information about the parameters and environment variables that are passed to the applications.

Given the sizes of modern disks, we do not expect storage requirements to be an important issue in practice. Nevertheless, Mirage limits storage (and time) overheads by not explicitly logging the content of input files and triggers trace collection only when necessary.

**Validation subsystem.** When an upgrade is received and applied locally, the system can determine which files were changed and, by inspecting the dependency information, which applications are affected by the upgrade.

To test an upgrade, during idle periods at the user machine, the upgrade-validation subsystem applies it to each affected application, feeds the upgraded application with its traced inputs, and compares the outputs to the traced outputs. Testing is performed for every set of inputs and outputs collected for each application. The test may fail because the upgrade does not integrate properly with an application, makes it crash when run with the traced inputs, or produces different outputs (upgrades of applications without traces can only be checked automatically for integration and crashing problems).

Performing upgrades first in an isolated environment, e.g., a virtual machine, allows for checking the (upgraded) applications' behavior before the upgrade is actually integrated into the production system, thereby simplifying rollback. To ensure that the upgraded applications behave the same in isolation as they would in the production system, the two environments are built from the same file system state. For this reason reads of files do not have to be explicitly recorded. We currently use a version of User-Mode Linux (UML) [34] as our virtual machine. We modify it to boot directly from the host's file system using copy-on-write.

The applications upgraded in the virtual machine are started and fed the recorded inputs by the UML kernel. To allow for non-determinism in the trace replay, Mirage maps the

recorded file inputs to the appropriate file operations, even if they are executed in a different order than in the trace. To avoid producing side-effects on remote systems even though the upgraded application is running in a virtual machine, Mirage silently drops the network outputs produced by the upgraded application, i.e., each write to a network socket succeeds but no data is actually sent out. Obviously, Mirage records these dropped outputs locally for comparison with the traced outputs. When outputs do differ, the user of the machine is asked to determine whether the observed behavior is acceptable for the application. This decision can be based on the differences between expected and actual outputs. Upon testing failure, the user can discard the upgrade by simply throwing away the virtual machine state. As part of our future work, we intend to investigate ways to provide a comparison context that simplifies the user’s decision.

### 3.4 Reporting Subsystem

After the success or failure of an upgrade is determined, the reporting component provides feedback to the vendor. Using the information about the context in which the problems are found in the field, the vendor can more easily pinpoint the reasons for misbehavior and fix its upgrades.

The collection of success/failure reports from all machines and clusters is transferred to an *Upgrade Report Repository (URR)* that can be queried by the vendor. The URR stores the following: (1) information about the cluster of deployment; (2) succinct success/failure results of upgrade tests; and (3) *report images* that allow the vendor to reproduce the upgrade problems. The storage that makes up this repository can be distributed or centralized. For example, sensitive or proprietary information about an installation may require local storage at the installation (and an explicit access permission by others) for privacy concerns. For security reasons, result reporting should use encryption and verifiable metadata to avoid snooping and attacks.

Our current implementation always co-locates the URR with the vendor. Further, each of our report images currently contains the entire upgraded virtual machine state, including recorded inputs and outputs used during replay.

### 3.5 Discussion and Current Limitations

Although the presentation above focuses on vendors and individual users, Mirage can benefit the administrators of large systems as well. In fact, a large system with uniform machine configurations is represented as a single machine in Mirage’s deployment protocol. Even when system administrators are required to certify all upgrades manually before deploying them, Mirage can help them by keeping track of the applications that need to be tested, automatically testing each of them, and reporting unexpected behavior. Furthermore, by increasing the administrators’ confidence in the upgrades, Mirage encourages administrators to apply upgrades sooner. In addition, by automating the reporting of failed upgrades, Mirage makes it easier for vendors to fix the problems with their upgrades.

Despite its many potential benefits, the current implementation of Mirage has some limitations.

**Deployment.** The information that the vendor stores about the similarities between the user machines could be used by an attacker to quickly identify the targets of a known vulnerability. Although items contain hashes of the environmental resources’ contents, the file names are currently stored in

the clear. Computing a cryptographic hash of file names would not help, as the attacker could install the application and check for identical hashes of vulnerable files. However, we could enhance the security and privacy of our “original”, parser-aided clustering, by letting each machine determine the cluster it belongs to locally based just on the comparison with the vendor’s set of files and fingerprints and by communicating a single cryptographic hash of its items to the vendor. To enable staging, the vendor could transmit only the list of clusters’ hashes to each machine. If necessary, machines could use some of the anonymity-preserving techniques [6] to communicate with the vendor. During deployment, the vendor could publicly advertise the cluster being tested currently and use only the number of nodes that belong to each cluster to determine when it is appropriate to move to the next stage.

**User-Machine testing.** Our current user-machine testing implementation has a number of limitations, many of them in common with other testing methods based on I/O comparison during replay. Among others, it does not guarantee correctness, but only identical behavior on test inputs; it compares the content of I/O only, not the performance of the application; and it has difficulty in dealing with certain forms of non-determinism. One limitation specific to our approach is that, while it handles many types of upgrades, it cannot be used for upgrades of operating system drivers, because we isolate the applications during validation.

Some problem-free upgrades change the I/O behavior of applications. In particular, upgrades that introduce new features may lead to different I/O and consequently fail testing. Mirage can be used to handle those upgrades by producing traces of the affected applications’ runs at the representatives, after they have approved the upgrade. These traces can be sent to the other members of the cluster to allow them to test the new features locally without human intervention. Therefore, Mirage can be used to handle major version upgrades, and not just “dot releases” or small patches.

We emphasize that flawless user-machine testing is not mandatory for the effectiveness of our infrastructure. In fact, we believe that it is unlikely that one could ever achieve perfect, fully-automatic user-machine testing. Consequently, a human (administrator, operator, beta tester, or user) may have to be involved in some cases. Staged deployment seeks to reduce the effort required of humans as much as possible.

Further, our user-machine testing component can be easily replaced by a component with similar functionality, without affecting the deployment and reporting subsystems at all. For example, to improve the quality of user-machine testing, we can incorporate a number of existing “white-box” upgrade-testing techniques based on source code analysis, e.g., [35]. Evaluating the effectiveness of our user-machine testing approach is important, but is beyond the scope of this paper.

**Problem reporting.** Our current implementation of problem reporting does not address privacy or security issues. However, for these purposes, we can leverage the same techniques currently used by commercial software vendors, such as by Microsoft’s Online Crash Analysis [21]. Furthermore, our reporting infrastructure can be easily replaced without affecting the user-machine testing and deployment components at all. Again, an evaluation of problem reporting approaches is beyond the scope of this paper.



App.	Files total	Env. resources	False positives	False negatives	Required vendor rules
firefox	907	839	1	23	7
apache	400	251	133	0	2
php	215	206	0	0	0
mysql	286	250	0	33	1

Table 1: Effectiveness of the heuristic in identifying environmental resources.

## 4. EVALUATION

We address the following three questions:

1. Can we accurately identify the environmental resources on which an application depends?
2. Can we cluster machines into meaningful clusters such that members of a cluster behave similarly or identically with respect to an upgrade?
3. Can we use staged deployment to significantly reduce upgrade overhead with little or no impact on deployment latency?

### 4.1 Identifying Environmental Resources

We study a desktop application (Firefox) and three server applications (Apache, PHP, and MySQL). To determine false positives (files the heuristic erroneously flags as environmental resources) and false negatives (environmental resources the heuristic missed), we manually inspect the set of files produced by the heuristic, as well as their contents.

Table 1 shows, for each application, the number of files accessed in the traces, the number of environmental resources, the number of false positives and false negatives generated by the heuristic, and the number of vendor-specified rules necessary to obtain a perfect classification of the files.

The heuristic by itself is able to correctly classify a large fraction of the environmental resources. Combining the heuristic with a small number of vendor-specified rules correctly identifies all environmental resources, without false positives or false negatives.

For PHP, the heuristic correctly identifies all environmental resources, without false positives or false negatives.

For Apache, the heuristic erroneously classifies the access log and some HTML files from the root document directory as environmental resources – the log because it is accessed during initialization, and the HTML files because they are accessed read-only in each run.

For MySQL, the heuristic erroneously omitted the directory in which the `mysql` database is stored, because by default it excludes files from `/var`. These files, however, also contain configuration data and are part of the environmental resources of MySQL.

For Firefox, not all extension, theme, and font files are included properly by the heuristic, because Firefox loads them whenever they are needed (i.e., after the initialization phase). We had to specify the important file types so they are correctly classified.

As can be seen from the table, the number of files misclassified can vary significantly from one application to the other. The number of rules, however, stays very small because each of these rules recognizes one special case of misclassification under which a variable number of files may fall. For this reason, the number of rules required to obtain a perfect classification gives a better appreciation of the effectiveness of the heuristic in terms of its ability to classify files automatically.

Defining the rules is an easy matter of invoking the regular expression-based Mirage API. In general, however, some files and directories are located at different places on different machines. In this case, the vendor can easily provide a script to automatically extract the correct location of files and directories from relevant configuration files or environment variables and generate the regular expressions locally on each machine.

### 4.2 Clustering

As mentioned in Section 3.2.3, a clustering is evaluated relative to a particular upgrade and a set of problems. We start by introducing metrics that quantify the “goodness” of clustering. The first metric captures the number of unnecessarily created clusters ( $C$ ), while the second metric captures the number of wrongly-placed machines ( $w$ ), i.e., machines that behave differently than the rest of their clusters.

If there are  $p$  different problems, an *ideal* clustering creates  $p + 1$  clusters (a cluster for every problem, and a cluster containing all other machines), has no unnecessary clusters ( $C = 0$ ), and has no misplaced machines ( $w = 0$ ). A *sound* clustering has  $C \geq 0$  and  $w = 0$ . With a sound clustering, therefore, multiple clusters may exhibit correct behavior or the same incorrect behavior. A clustering that is *imperfect* has  $w > 0$  and potentially  $C > 0$ .

Unfortunately, an ideal clustering is typically not achievable in practice. Since a particular problem may affect only a subset of environments, there may be multiple clusters with machines that do not experience the problem. In contrast, sound clustering is achievable and useful in limiting the upgrade overhead, so it is the type of clustering that we seek to obtain.

We have reproduced a number of machine environments that exhibit real, non-trivial upgrade problems with two applications: MySQL and Firefox.

#### 4.2.1 MySQL

Table 2 describes the machine configurations used in the MySQL experiment. On some of these configurations, the upgrade to a new version is successful. Other configurations (the ones with `php4` in their names) are known to exhibit broken-dependency problems with PHP [24] after a MySQL upgrade. Still others (the ones with `userconfig` in their name) exhibit legacy-configuration problems with the `my.cnf` MySQL configuration file.

Figure 6 shows the results of our clustering algorithm when we use application-specific parsers for *all* environmental resources. Entries in boldface denote machines that experience the PHP problem when upgrading MySQL, whereas the entries in bold-italics denote machines that experience the `my.cnf` problem.

As we see from Figure 6, the machines that experience the MySQL upgrade problems are not clustered together with machines experiencing other problems or machines behaving

Machine Name	Description
fc5-ms4	Fedora Core 5
fc5-ms4/php4	Fedora Core 5 with PHP 4.4.6
fc5-ms4/php4/ap139	Fedora Core 5 with PHP 4.4.6 and Apache 1.3.9
fc5-ms4/php4-comments	Fedora Core 5, PHP 4.4.6 (comments in <code>/etc/mysql/my.cnf</code> altered)
ubt-ms4, ubt-ms4(2)	Ubuntu 6.06 (Dapper Drake) (two identical machines)
ubt-ms4/php4	Ubuntu 6.06 with PHP 4.4.6
ubt-ms4/php4/ap139	Ubuntu 6.06 with PHP 4.4.6, and Apache 1.3.9 (compiled with PHP support)
ubt-ms4/withconfig	Ubuntu 6.06 (added config. file <code>/etc/mysql/my.cnf</code> )
ubt-ms4/userconfig	Ubuntu 6.06 (added a user config file <code>\$HOME/.my.cnf</code> )
ubt-ms4/confdirective-added	Ubuntu 6.06 (added a config. directive to <code>/etc/mysql/my.cnf</code> )
ubt-ms4/confdirective-deleted	Ubuntu 6.06 (deleted a config. directive from <code>/etc/mysql/my.cnf</code> )
ubt-ms4/comment-added	Ubuntu 6.06 (added some comments to <code>/etc/mysql/my.cnf</code> )
ubt-ms4/comment-deleted	Ubuntu 6.06 (deleted some comments from <code>/etc/mysql/my.cnf</code> )
ubt-ms4/libc-upg	Ubuntu 6.06 (upgraded to a new version of libc)
ubt-ms4/libc-upg/withconfig	Ubuntu 6.06 (added config. file <code>/etc/mysql/my.cnf</code> , libc upgraded)
ubt-ms4/libc-upg/userconfig	Ubuntu 6.06 (added a user config. file <code>\$HOME/.my.cnf</code> , libc upgraded)
ubt-ms4/libc-upg/confdirective-added	Ubuntu 6.06 (added a config. directive to <code>/etc/mysql/my.cnf</code> , libc upgraded)
ubt-ms4/libc-upg/confdirective-deleted	Ubuntu 6.06 (deleted a config. directive from <code>/etc/mysql/my.cnf</code> , libc upgraded)
ubt-ms4/libc-upg/comment-added	Ubuntu 6.06 (added comments to <code>/etc/mysql/my.cnf</code> , libc upgraded)
ubt-ms4/libc-upg/comment-deleted	Ubuntu 6.06 (deleted comments from <code>/etc/mysql/my.cnf</code> , libc upgraded)

**Table 2: Sample Linux configurations used in our experiments. All machines have MySQL 4.1.22.**

correctly, thus satisfying our main clustering goal ( $w = 0$ ). During deployment, problems would be observed in only two clusters.

The results also show that the algorithm creates separate clusters for the different distributions, versions of libc, dependent applications, and differences in the configuration files. Although these differences have no bearing on the behavior of this particular upgrade, another upgrade could be problematic in one of these different environments. Thus, the algorithm must conservatively cluster these machines separately. Overall, this clustering created 12 additional clusters ( $C = 12$ ) with respect to this upgrade.

We now examine the results of the clustering algorithm without using any MySQL-specific parsers; the initial phase of the algorithm relies on Mirage-supplied parsers only. As described in Section 3.2.3, the Mirage-supplied parsers deal with executables, shared libraries, and system-wide configuration files. Therefore, we have to execute the second phase of the clustering algorithm, which uses the QT algorithm for the resources that are fingerprinted with content-delineation. We show the final clustering results obtained with a diameter value of 3 in Figure 7.

The figure shows that, even without the application-specific parsers, our clustering algorithm correctly clusters the machines with the PHP-related problem. However, we see that clusters 4 and 6 both contain a machine that experiences a problem along with machines that do not. This clustering is thus not sound ( $w = 2$ ) for an upgrade that triggers the `my.cnf` problem. This happens because the number of differences between the machines in this cluster is smaller than the diameter value. Specifically, the differences in this case are in the hash of the `my.cnf` file. Since the file is rather small and our Rabin fingerprinting takes hashes at the granularity of 4KB, only one hash is different. Using a diameter value of 0 would have separated the machines in this cluster, but it would also have separated machines that have benign differences (the machines with `-comment`

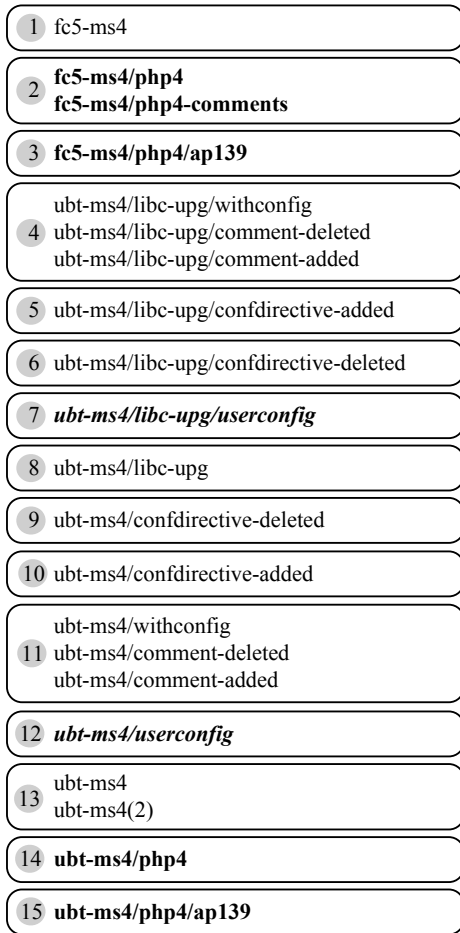
in the name). In a real-world scenario with thousands of machines, this choice of the diameter could result in a large number of clusters that would slow down the deployment. This example shows that picking an adequate diameter value is difficult and that clustering might be imperfect, regardless of the diameter value chosen, when some parsers are missing.

The vendor can decide to vary the size of the resulting clusters depending on the upgrade that is to be deployed. For instance, in our setup we have 5 different types of changes to the MySQL configuration file `my.cnf`. If the vendor is providing a parser for this file and wants to deploy an upgrade that is unlikely to be affected by any of those changes, it can choose to ignore the corresponding items. The resulting merge of clusters 4,5,6 and clusters 9,10,11 in Figure 6 can speed up the staged deployment. This re-grouping of clusters still correctly separates the problematic configurations from other, “good” clusters. Such an effect cannot easily be achieved if the vendor does not provide a parser. Without a parser, creating larger clusters can only be done by increasing the diameter, which does not give any guarantee over how machines are going to be re-clustered.

#### 4.2.2 Firefox

Table 3 describes the machine configurations used in the Firefox experiment. The three first configurations are fresh installations of version 1.5.0.7 of Firefox, with the third one having Java and JavaScript disabled. The other three configurations have been upgraded from version 1.0.4, and the last one has Java and JavaScript turned off. The latter three configurations exhibit a legacy-configuration problem when upgraded to Firefox 2.0. In particular, two preference files that existed in version 1.0.x (and were upgraded to 1.5.x) cause erratic behavior when Firefox is upgraded to version 2.0 [11].

Figure 8 shows results obtained using application-specific parsers for Firefox’s configuration files, in addition to Mi-

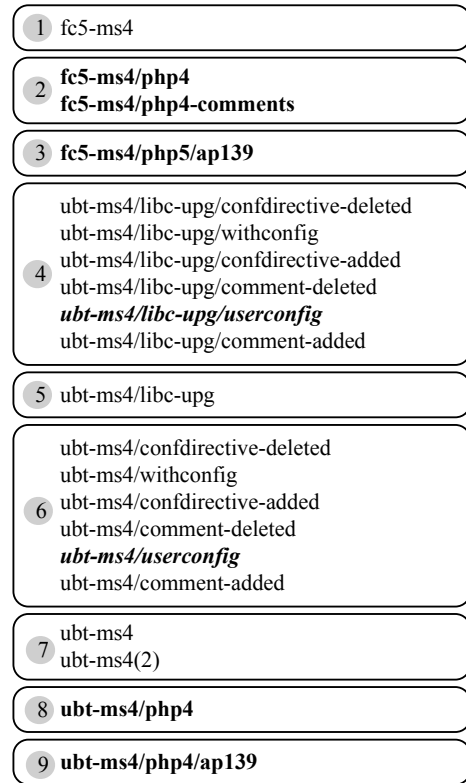


**Figure 6:** Clustering obtained by using parsers for all environmental resources. Entries in boldface experience the PHP problem with a MySQL upgrade, whereas entries in bold-italics will exhibit the my.cnf problem.

rage’s supplied parsers. The clustering is sound ( $w = 0$ ,  $C = 2$ ) for this upgrade; there are two extra clusters with the two machines with Java and JavaScript disabled.

Figure 9 shows two clustering results using only Mirage-supplied parsers. On the left, a diameter value of 4 is used. The problematic configurations are all in one cluster, and all the other ones in another cluster. For the upgrade we are considering, this clustering is ideal, with  $w = 0$  and  $C = 0$ . On the right, a diameter value of 6 is used. This clustering is imperfect, as the problematic configurations are clustered with other machines ( $w = 3$ ). These results show that picking the right clustering diameter is difficult and that a small difference potentially has a non-trivial impact.

The comparison between the clustering results in Figures 8 and 9(left) might suggest that it is sometimes better for the vendor not to provide parsers for its applications. However, this conclusion is incorrect. The reason is that, when the vendor is about to deploy an upgrade, it does not know if or where problems may occur. Thus, it would be improper to assume, as done in Figure 9(left), that machines with features turned off always behave the same as their counterparts with those features turned on. Firefox’s configuration files contain many irrelevant parameters (such



**Figure 7:** Clustering obtained by using only parsers supplied by Mirage and a diameter value of 3 for the remaining environmental resources. Entries in boldface experience the PHP problem, whereas entries in bold-italics will exhibit the my.cnf problem.

as timestamps and window coordinates) as well as relevant configuration settings (such as Java settings). Since diameter clustering only considers the number of differences, it is unable to make a distinction between relevant and irrelevant differences. If the vendor supplies the required parsers, it can discard the irrelevant differences to guarantee sound clustering.

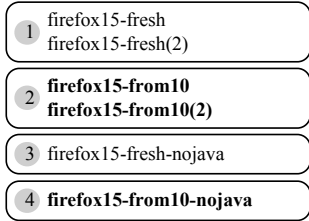
### 4.3 Deployment Protocols

We still need to evaluate the impact of staged deployment on upgrade overhead and latency. Toward this end, out of the space of possible staged deployment protocols described in Section 3, in this section we evaluate two: one that seeks to front-load most of the vendor’s debugging effort while reducing the upgrade overhead (i.e., the number of machines testing a faulty upgrade); and one that seeks to reduce upgrade overhead without disregarding the upgrade latency (i.e., the time elapsed until the upgrade is successfully applied and running on a machine). For both protocols, we assume that the representatives are always online and ready to fully test an upgrade, perhaps as a result of a financial arrangement with the vendor.

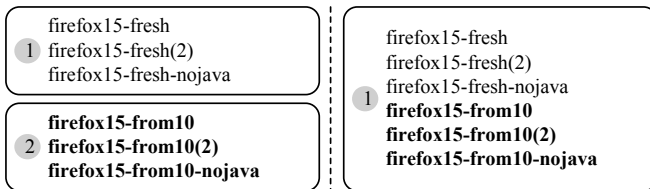
**Front-loading the debugging effort while reducing the upgrade overhead.** The first protocol (called Front-Loading, for short) is divided into two phases. In the first phase, the vendor starts by notifying all representatives of all clusters in parallel that an upgrade is available. The

Machine Name	Description
firefox15-fresh, firefox15-fresh(2)	Firefox v.1.5.0.7 freshly installed (two identical machines)
firefox15-fresh-nojava	Firefox v.1.5.0.7 freshly installed, Java and JavaScript disabled
firefox15-from10, firefox15-from10(2)	Firefox v.1.5.0.7, upgraded from v.1.0.4 (two identical machines)
firefox15-from10-nojava	Firefox v.1.5.0.7, upgraded from v.1.0.4, Java and JavaScript disabled

**Table 3: Configurations used in our experiments with Firefox. All machines run the same Linux distribution.**



**Figure 8: Firefox clustering obtained by using parsers for all environmental resources. Entries in boldface experience problems when upgrading to version 2.0.**



**Figure 9: Firefox clustering results obtained by using only Mirage-supplied parsers. On the left, results with a diameter value of 4. On the right, results with a diameter value of 6. Entries in boldface experience problems when upgrading to version 2.0.**

representatives then download and test the upgrade using Mirage’s user-machine testing subsystem (or another appropriate testing approach). Any problems during this phase are reported back to the vendor using Mirage’s reporting subsystem. Once the problems are fixed by the vendor, all representatives are again concurrently notified about the corrected upgrade and can download, test, and report on it. This process is repeated until no more problems are reported by the representatives, ending the first phase of the protocol. No non-representatives test before the end of this phase.

During the second phase, the vendor proceeds by handling one cluster at a time, sequentially, starting from the cluster that is most dissimilar to the vendor’s environment and proceeding (in reverse order of similarity) toward the cluster that is most similar to it. Ideally, no new problem should be discovered in this phase. However, because of imperfect testing or clustering, some problems may still be encountered. The vendor starts this phase by simultaneously notifying the non-representative machines of the first cluster in the order. The non-representatives of this cluster then download the upgrade, test it, and report their results. It is possible that the upgrade succeeds at some non-representative machines but fail at others. When this occurs, the non-representative machines that passed testing are allowed to integrate the upgrade, but are later notified

of a new upgrade fixing the problems with the original upgrade. The machines that failed testing do not integrate the upgrade and are later notified of a (hopefully) corrected version of the upgrade. When there are no more failures and a large fraction (according to a vendor-defined threshold) of the non-representatives have passed upgrade testing successfully, the process is repeated for the next cluster in the order. The reason we only wait for a (large) fraction of the non-representative machines to report success is that some machines may stay offline for long periods of time; it would be impractical to wait for all these machines to pass testing before moving to the next cluster. When these “late arrivals” come back online, they are notified, download, test, and report on all the upgrades they missed.

This protocol front-loads the debugging effort in two ways: (1) it collects reports from all representatives of all clusters right from the start, getting a broad picture of all the problems that can be expected; and (2) in its second phase, it proceeds in the reverse order of cluster distance from the vendor, as farther clusters are more likely to experience problems. These two characteristics also allow the protocol to reduce the upgrade overhead.

**Reducing upgrade overhead with better upgrade latency.** In our second staged protocol (called *Balanced*, for short), deployment starts with the cluster that most closely resembles the vendor’s installation, progresses sequentially to the next cluster that most closely resembles the vendor, and so on. The similarity between clusters and the vendor is evaluated by a distance function accounting for the number of differences in the environmental resources of machines belonging to the clusters. Each time the protocol progresses to a new cluster, the representatives of that cluster are notified about it in parallel and can then download, test, and report on it. This process is repeated until the upgrade passes testing successfully at all the representatives of the cluster. At this point, the non-representative machines of the cluster are notified and go through the same process. As in *Front-Loading*, non-representatives that pass testing successfully can integrate the upgrade, but may later receive a new upgrade. When all (or a significant fraction of) the machines in a cluster have successfully tested an upgrade, deployment progresses to the next cluster.

This protocol promotes low upgrade overhead by testing upgrades at the representatives of each cluster before allowing the much larger number of non-representatives to test them. Interestingly, it also promotes low latency for a large set of machines, as compared to our first protocol, in two ways: (1) by allowing many non-representatives to successfully pass testing before problems with all clusters are debugged by the vendor; and (2) by ordering clusters so that clusters that are more likely to pass testing receive the upgrades first. However, debugging at the vendor is more spread out in time, as compared to the first protocol.

### 4.3.1 Evaluation Methodology

The evaluation is based on an event-driven simulator of different deployment protocols and clustering schemes. The main simulator inputs are the number of clusters, the clusters’ sizes, the clustering quality, the number of representatives per cluster, in which clusters the upgrade problems appear, and the times to download, test, and fix an upgrade. Because our goal here is not to present an extensive set of results for a large number of simulator inputs, we select a particular scenario and discuss the corresponding results. The behaviors that we observe with this example scenario are representative of those of the other scenarios we considered.

Our example scenario has 100,000 simulated machines running Mirage. The times to download and test an upgrade are 5 and 10 time units, respectively. The time to fix each upgrade problem at the vendor is set at 500 time units. We choose the ratio between upgrade download and test vs. debugging time to mimic the ratio between the expected times in the field; download and test taking a few tens of minutes, with debugging (entire cycle at the vendor) taking at least one day. In all cases, we cluster machines into 20 clusters.

We consider two main categories of upgrade problems: prevalent (a large number of machines are affected by the problem) and non-prevalent (a smaller number of similar machines are affected). The results we show are for a case of one prevalent problem affecting 15% of machines (resembling the upgrade failure results reported in the literature [4]) and two non-prevalent problems in two different clusters.

We use two different clusterings. The first is sound with respect to the problems we defined; it has 16 more clusters than ideal clustering, with no misplaced machines. We create the second, imperfect, clustering by injecting a single misplaced machine in one of the clusters from the sound clustering setup. This machine is not a representative in the affected cluster. We assume that user-machine testing correctly identifies a problem if there is one, with no false positives. Hence, we assign one representative per cluster.

Given the lack of large-scale machine characterizations that we would require in this scenario, we assume that all clusters have the same size and study per-cluster, rather than per-machine, upgrade latencies. In the cases we consider, the vast majority of machines behave the same within a cluster. Hence, this setup enables us to discuss the qualitative features of our staged deployment protocols.

We compare our two protocols, FrontLoading and Balanced, against baselines called “NoStaging” and “RandomStaging”. The NoStaging protocol places all machines into a single cluster and treats them all as representatives to promote deployment speed at the possible cost of a high upgrade overhead. For this reason, NoStaging should be used for simple and urgent upgrades, such as security patches. The RandomStaging protocol resembles Balanced, but the order of deployment to clusters is random. Although RandomStaging may not have any real use in practice, it does allow us to isolate the benefit of staged deployment from that of intelligent machine clustering. To evaluate this protocol, we create a scenario in which the problems are uniformly distributed across the deployment order.

Since we do not have an underlying set of machine environments, we consider two deployment scenarios for the

Balanced protocol: 1) “best-case”, in which the representatives discover problems at the latest possible time, producing the most favorable latency; and 2) “worst-case”, with the problems being encountered early on, resulting in the worst latency. We compare the protocols in terms of their latencies and upgrade overheads.

### 4.3.2 Deployment Results

Most importantly, we expect our staged protocols to have significantly lower upgrade overhead than NoStaging. This benefit should come with little or no penalty in terms of upgrade latency.

**Sound clustering.** Our protocols indeed exhibit much lower upgrade overhead than NoStaging. With NoStaging, all machines that exhibit problems ( $m$ , in total) fail testing, whereas many fewer do so with the other three protocols thanks to their staged deployment (presumably,  $m \gg p$ , i.e., the number of problematic machines is much greater than the number of problems,  $p$ ). The main reason for this result is that staging forces the deployment to halt at the first cluster that exhibits a problem, sparing other machines from experiencing the same problem. Since the clustering is sound with no misplaced machines and we assume that user-machine testing finds the problem if one exists, the representative of each cluster with a problem experiences it. Hence, the total number of test failures is  $p$  for Balanced and RandomStaging. As expected, Balanced has lower overhead than FrontLoading. With FrontLoading, representatives of all clusters with machines exhibiting the prevalent problem ( $p + C_p$  in total, where  $C_p$  is the number of additional clusters with problematic machines) fail testing.

Figure 10 shows the CDF of the upgrade latency, confirming our intuition about the protocols we consider. In NoStaging, 75% of the machines pass the upgrade test right away (at the cost of the high upgrade overhead we just discussed). In the best-case scenario, Balanced quickly applies the upgrade successfully at a large fraction of machines, significantly sooner than FrontLoading. Since FrontLoading has an initial parallel testing and debugging phase, most successful upgrades are delayed by several debugging times at the vendor (500 time units each). Nevertheless, the last cluster applies the upgrade sooner under FrontLoading than the other staged protocols, especially when the number of clusters is large, since FrontLoading’s second phase does not have an extra step of testing at representatives. RandomStaging’s latency lies between that of the best-case and the worst-case scenarios for Balanced. We also see that additional clusters (when  $C > 0$ ) slow down the sequential phases of the staged protocols.

**Imperfect clustering.** Figure 11 shows the CDF of the per-cluster latency when we position the misplaced machine (a problematic machine that behaves differently from the rest of its cluster) in the first or the last cluster in the order.

We see that FrontLoading can be further slowed down if the misplaced machine is in the first cluster. Arguably, this is in agreement with this protocol’s main strategy. With its best-case scenario, Balanced is similarly slowed down in this case when the entire first cluster tests the upgrade and we encounter the misplaced machine. When the misplaced machine is in the last cluster in the order, this protocol is only marginally affected. Since it tests on all machines, NoStaging is not affected by the presence of the misplaced machine. Overall, the main trends among the protocols remain and

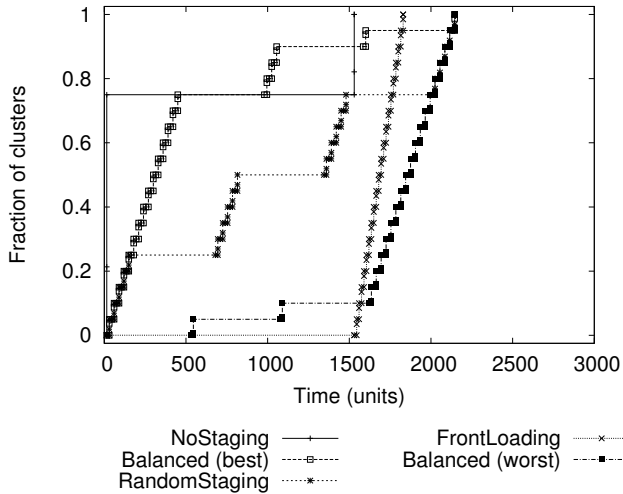


Figure 10: CDF of the per-cluster upgrade latency under sound clustering.

the upgrade overhead is simply augmented by one extra machine that fails testing for all protocols.

## 5. RELATED WORK

### 5.1 Characterizing Upgrades

As far as we know, only two other works characterized upgrades. Beattie *et al.* [4] tried to determine when it is best to apply security patches. They built mathematical models of the factors affecting when to patch, and collected empirical data to validate the model. Relative to Mirage, the paper focused solely on security patches and did not consider the benefits of user-machine testing or staged deployment.

The study by Gkantsidis *et al.* [13] focused on the Windows environment and on the networking aspects of deploying upgrades. Most importantly, the study did not consider several important issues, including upgrade installation and testing, upgrade problems, and problem reporting. Our survey addressed all of these issues.

### 5.2 Deployment Subsystem

**Upgrade deployment.** As far as we know, no previous work has considered staged upgrade deployment. In fact, the two previous works on large-scale upgrade deployment focused solely on (1) deploying upgrades as quickly as possible to all users while reducing the load on the vendor’s site [13]; and (2) creating a deployment infrastructure that can be tailored by explicit contracts between vendors and users [31]. In contrast to (1), Mirage recognizes the tradeoff between the speed of deployment and the likelihood of testing problems at the user machines. In contrast to (2), Mirage does not consider contracts, since they are rare in practice, especially in the open-source community.

Commercial upgrade (patch) management systems and tools, e.g., [17, 27], deploy upgrades within enterprises. They typically concentrate on discovering machines on the enterprise’s network, assessing which upgrades are needed, and applying the patches while minimizing the number of re-

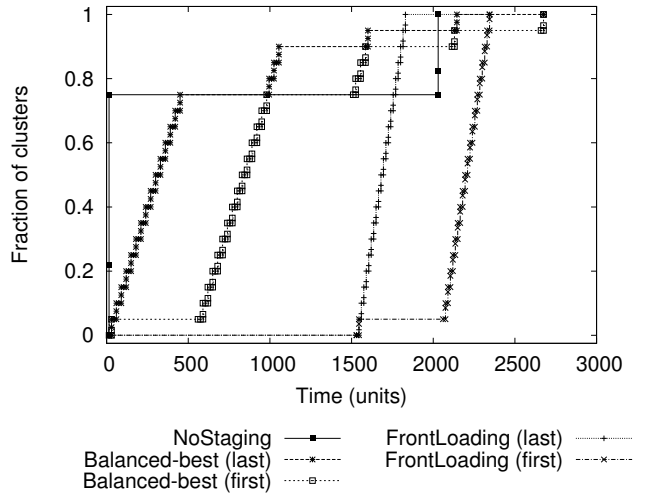


Figure 11: CDF of the upgrade latency under imperfect clustering. The position of the misplaced machine (its cluster in the deployment order) is shown in parenthesis after protocol names.

boots. Some providers, such as [27], manually collect upgrades from vendors and test them locally before sending them to their customer enterprises. Relative to Mirage, these upgrade-management systems do not help individual users and have no automatic support for testing upgrades after they are applied to the machines in the enterprise. Nevertheless, the providers that manually test upgrades do produce a primitive form of staging for enterprises. However, they are unlike representatives (or beta testers), since they have no relationship to the vendor. In fact, their existence does not change the fact that vendors deploy their upgrades to the entire world as a single huge cluster.

**Clustering machines.** The Pastiche [8] peer-to-peer backup system builds a structured overlay by replacing the proximity metric with a measure of similarity between sets of neighbors’ files. Although Pastiche does not explicitly cluster machines, one could envisage Pastiche being used to perform distributed clustering based on the list of environmental resources at each machine.

Others have considered surviving Internet catastrophes by backing up data on dissimilar hosts. This problem roughly corresponds to the problem of recovering from a globally-deployed problematic upgrade. In Phoenix [16], Junqueira *et al.* observe that backups should be done on hosts that do not share the same potential vulnerabilities. Interestingly, the clustering of dissimilar hosts in Phoenix has the opposite goal to our clusters, which cluster similar hosts. Furthermore, Mirage clusters machines based on different characteristics than Phoenix.

**Integrating deployment, testing, and reporting.** At a high-level, the most similar work to Mirage is a position paper by Cook and Orso [7], which also proposes to integrate upgrade deployment, user-machine testing (via white-box approaches), and problem reporting back to developers. However, they did not consider staged deployment or any type of clustering. Furthermore, they did not attempt to characterize upgrades or evaluate their proposed system.



### 5.3 User-machine Testing Subsystem

**Dependency tracking.** When installing new software, e.g., an upgraded application, package-management systems (such as RedHat's yum) provide a convenient way to find and install all the required packages. However, unlike Mirage, package-management systems do not address upgrade testing or reporting activities. Mirage prevents a larger set of upgrade problems while helping vendors debug their upgrades based on feedback from their users' machines. Nevertheless, Mirage can benefit from the detailed version information provided by package-management systems to identify the source of installation problems.

The Vesta [14] software configuration management tool tracks file dependencies to reduce the build time. Unlike Mirage, dependency tracking is done at the file-system level by monitoring accesses to the Vesta repository. Mirage can track dependencies in any live system, while Vesta works only for source files and build processes.

BackTracker [18] and PASS [22] use system call interception to track dependencies for security and file provenance purposes, respectively. Mirage does so to determine when to re-start tracing and what applications to test when a new upgrade is received.

The Strider project at Microsoft Research produced the Patch Impact Analyzer [9]. This tool dynamically determines application dependency information, performs pre-upgrade impact analysis, and checks whether DLLs are loaded after a patch is applied. This tool was rolled into the Application Compatibility Toolkit, enabling users to test compatibility with a version of Windows, and in particular with the one the user is running. In contrast, Mirage is more general, as it relies on external output to determine whether an upgraded application works properly.

**Replay and comparison.** Previous works [25, 26] have considered validating operator actions (including software upgrades) by replaying requests and checking replies in a custom, isolated extension of Internet services. Mirage uses an off-the-shelf virtual machine to test upgrades to any type of software (not just servers), instead of a custom environment. Agrawal and Seshan [1] also mention the use of virtual machines for upgrading distributed applications, but leave other aspects of the approach for future work. For a distributed application, the main concern has typically been to perform the upgrade while the application continues to execute [2]. We have not considered distributed applications so far.

Other work has considered application tracing and deterministic replay for debugging purposes, e.g. [10, 29, 32]. These systems are substantially more complex than Mirage, as they have to checkpoint internal states and replay the details of an execution exactly.

Relative to approaches in comparing descriptions of component and other behavior [5, 19, 20, 35], Mirage does not rely on any description, specification, or additional effort at development time for user-machine testing. Furthermore, it treats applications as black boxes for which users may not have source codes. Nevertheless, Mirage can be retargeted to use white-box approaches for user-machine testing without affecting its deployment component at all.

### 5.4 Reporting Subsystem

Today, software vendors routinely include software crash reporting facilities in their products. For example, when an

application crashes in Microsoft Windows, it is possible to send an error report back to Microsoft [21]. This report may not include enough information to be easily reproduced, requiring significant human labor in debugging. Perhaps most importantly, no reports are transferred when an upgrade causes unexpected/unacceptable behavior without crashing or when the upgrade causes other applications (that share resources with the upgraded application) to misbehave. In contrast, the Mirage reports are sent whenever an upgrade fails testing, due to crashing or not.

## 6. CONCLUSION

We introduce Mirage, a distributed framework that integrates deployment, user-machine testing, and problem reporting into the development cycle of software upgrades. The design of Mirage is motivated by a survey of system administrators that points to a high incidence of problematic upgrades, and to the difference between vendor and user environments as one of the principal causes of upgrade problems. We describe in detail two key innovations in Mirage: the clustering of user machines according to their environments and the staged deployment of upgrades to these clusters.

Our evaluation demonstrates that our clustering algorithm achieves its goal of separating machines with dissimilar environments in different clusters, while not creating an impractically large number of clusters. The algorithm performs substantially better when the vendor provides parsers for all of the environmental resources of the application. A simulation study of two staged deployment protocols demonstrates that staging significantly reduces upgrade overhead, while still achieving low deployment latency. Furthermore, a suitable choice of protocol allows a vendor to achieve different objectives.

## Acknowledgments

We would like to thank George Candea, Christopher Iu, and Steven Dropho for their useful comments. In addition, we thank our shepherd Rich Draves and our anonymous reviewers who provided excellent feedback. Olivier Crameri was awarded an SOSp student travel scholarship, supported by Infosys, to present this paper at the conference.

## 7. REFERENCES

- [1] M. Agrawal and S. Seshan. Development Tools for Distributed Applications. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.
- [2] S. Ajmani, B. Liskov, and L. Shrira. Scheduling and Simulation: How to Upgrade Distributed Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems*, May 2003.
- [3] ASF Bugzilla Bug 10073 upgrade from 1.3.24 to 1.3.26 breaks include directive. [http://issues.apache.org/bugzilla/show\\_bug.cgi?id=10073](http://issues.apache.org/bugzilla/show_bug.cgi?id=10073).
- [4] S. Beattie, S. Arnold, C. Cowan, P. Wagle, C. Wright, and A. Shostack. Timing the Application of Security Patches for Optimal Uptime. In *Proceedings of the 16th Systems Administration Conference*, 2002.
- [5] P. Brada. Metadata Support for Safe Component Upgrades. In *Proceedings of the 26th International Computer Software and Applications Conference*, August 2002.

- [6] D. Chaum. Untraceable Electronic Mail, Return Addresses, and Digital Pseudonyms. *Communications of the ACM*, 4(2), February 1981.
- [7] J. Cook and A. Orso. MonDe: Safe Updating through Monitored Deployment of New Component Versions. In *Proceedings of the 6th Workshop on Program Analysis for Software Tools and Engineering (PASTE)*, September 2005.
- [8] L. P. Cox, C. D. Murray, and B. D. Noble. Pastiche: Making Backup Cheap and Easy. In *Proceedings of the 5th Symposium on Operating Systems Design and Implementation*, December 2002.
- [9] 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 *Proceedings of the International Conference on Autonomic Computing*, May 2004.
- [10] G. W. Dunlap, S. T. King, S. Cinar, M. A. Basrai, and P. M. Chen. ReVirt: Enabling Intrusion Analysis Through Virtual-Machine Logging and Replay. *SIGOPS Oper. Syst. Rev.*, 36(SI):211–224, 2002.
- [11] Fixing A Troubled Firefox 2.0 Upgrade. <http://softwaregadgets.gridspace.net/2006/10/30/fixing-a-troubled-firefox-20-upgrade/>.
- [12] Firefox crashes after 1.5.0.9 update. <http://www.ubuntuforums.org/showthread.php?t=331274>.
- [13] C. Gkantsidis, T. Karagiannis, P. Rodriguez, and M. Vojnovic. Planet Scale Software Updates. In *Proceedings of SIGCOMM*, September 2006.
- [14] A. Heydon, R. Levin, T. Mann, and Y. Yu. *The Vesta Software Configuration Management System*. Compaq Systems Research Center, 2002.
- [15] L. J. Heyer, S. Kruglyak, and S. Yooseph. Exploring Expression Data: Identification and Analysis of Coexpressed Genes. In *Genome Research*, pages 1106–1115, 1999.
- [16] F. Junqueira, R. Bhagwan, A. Hevia, K. Marzullo, and G. M. Voelker. Surviving Internet Catastrophes. In *Proceedings of the USENIX 2005 Annual Technical Conference*, April 2005.
- [17] Kaseya Patch Management. <http://www.kaseya.com/products/patch-management.php>.
- [18] S. King and P. Chen. Backtracking Intrusions. In *Proceedings of the 19th SOSP*, October 2003.
- [19] L. Mariani and M. Pezze. Behavior Capture and Test: Automated Analysis of Component Integration. In *Proceedings of the International Conference on Engineering of Complex Computer Systems*, June 2005.
- [20] S. McCamant and M. D. Ernst. Predicting Problems Caused by Component Upgrades. In *Proceedings of the 10th European Software Engineering Conference and the 11th Symposium on the Foundations of Software Engineering*, September 2003.
- [21] Microsoft Online Crash Analysis. <http://oca.microsoft.com/en/Welcome.aspx>.
- [22] K.-K. Muniswamy-Reddy, D. A. Holland, U. Braun, and M. Seltzer. Provenance-Aware Storage Systems. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.
- [23] A. Muthitacharoen, B. Chen, and D. Mazieres. A Low-bandwidth Network File System. In *Proceedings of the 18th SOSP*, December 2001.
- [24] Report of PHP problem after MySQL upgrade. <http://www.linuxquestions.org/questions/showthread.php?t=425535>.
- [25] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the USENIX Symposium on Operating Systems Design and Implementation*, Dec. 2004.
- [26] F. Oliveira, K. Nagaraja, R. Bachwani, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Validating Database System Administration. In *Proceedings of the USENIX Annual Technical Conference*, June 2006.
- [27] PatchLink. <http://www.patchlink.com/>.
- [28] PHP5 Migration guide. <http://ch2.php.net/manual/en/migration5.incompatible.php>.
- [29] Y. Saito. Jockey: A User-Space Library for Record-Replay Debugging. In *Proceedings of the 6th International Symposium on Automated Analysis-Driven Debugging*, September 2005.
- [30] Secunia "Security Watchdog" Blog. <http://secunia.com/blog/11>.
- [31] L. Sobr and P. Tuma. SOFAnet: Middleware for Software Distribution over Internet. In *Proceedings of the IEEE Symposium on Applications and the Internet*, January 2005.
- [32] S. Srinivasan, S. Kandula, C. Andrews, and Y. Zhou. Flashback: A Lightweight Extension for Rollback and Deterministic Replay for Software Debugging. In *Proceedings of the USENIX Annual Technical Conference*, June 2004.
- [33] Software upgrade survey. <http://mirage.epfl.ch/webdav/site/mirage/users/128770/public/survey.pdf>.
- [34] User-Mode Linux. <http://user-mode-linux.sourceforge.net/>.
- [35] T. Xie and D. Notkin. Checking Inside the Black Box: Regression Testing Based on Value Spectra Differences. In *Proceedings of the 20th IEEE International Conference on Software Maintenance*, September 2004.