Carnegie Mellon University
Research Showcase

Parallel Data Laboratory | Research Centers and Institutes

12-1-2008

A Fault Model for Upgrades in Distributed Systems (CMU-PDL-08-115)

Tudor Dumitraş
Carnegie Mellon University

Soila Kavulya
Carnegie Mellon University

Priya Narasimhan
Carnegie Mellon University

Follow this and additional works at: http://repository.cmu.edu/pdl

Recommended Citation
Dumitraş, Tudor; Kavulya, Soila; and Narasimhan, Priya, "A Fault Model for Upgrades in Distributed Systems (CMU-PDL-08-115)" (2008). *Parallel Data Laboratory.* Paper 10.
http://repository.cmu.edu/pdl/10

This Technical Report is brought to you for free and open access by the Research Centers and Institutes at Research Showcase. It has been accepted for inclusion in Parallel Data Laboratory by an authorized administrator of Research Showcase. For more information, please contact research-showcase@andrew.cmu.edu.

A Fault Model for Upgrades in Distributed Systems

Tudor Dumitraş, Soila Kavulya and Priya Narasimhan

CMU-PDL-08-115
December 2008

Parallel Data Laboratory
Carnegie Mellon University
Pittsburgh, PA 15213-3890

Abstract

Recent studies, and a large body of anecdotal evidence, suggest that upgrades are unreliable and often end in failure, causing downtime and data-loss. While this is sometimes due to software defects in the new version, most upgradefailures are the result of faults in the upgrade procedure, such as broken dependencies. In this paper, we present data on upgrade failures from three independent sources — a user study, a survey and a field study — and, through statistical cluster analysis, we construct a novel fault model for upgrades in distributed systems. We identify four distinct types of faults: (1) simple configuration errors (e.g., typos); (2) semantic configuration errors (e.g., misunderstood effects of parameters); (3) broken environmental dependencies (e.g., incorrect libraries, port conflicts); and (4) complex procedural errors. We estimate that, on average, Type 1 faults occur in 15.2 % of upgrades, and Type 4 faults occur in 16.8 % of upgrades.

1 Introduction

Software upgrades are unavoidable in distributed enterprise systems. For example, business reasons sometimes mandate switching vendors; responding to customer expectations and conforming with government regulations can require new functionality. However, recent studies [3, 10, 17, 23, 34, 43] and a large body of anecdotal evidence [2, 9, 22, 27, 33, 38, 40] suggest that distributed systems suffer from frequent upgrade failures. A 2007 survey of 50 system administrators from multiple countries (82% of whom had more than five years of experience) identified broken dependencies and altered system-behavior as the leading causes of upgrade failure, followed by bugs in the new version, incompatibility with legacy configurations and improper packaging of the new components to be deployed [34].[1] This suggests that most upgrade failures are not due to software defects, but to *faults affecting the upgrading procedure*.

Upgrades in distributed systems usually require a complex change-management process that involves hardware and software additions, reconfigurations, and data migrations. The maze of dependencies in a distributed system includes relationships among various components, APIs, configuration settings, data objects, communication protocols, Internet routes or performance levels. When the old and new versions of the system-under-upgrade share dependencies (*e.g.*, they load the same dynamically-linked library but require different versions of its API), the upgrade might break some of these dependencies and induce downtime. Moreover, dependencies can sometimes remain *hidden*, because they cannot be detected automatically or because the are overlooked due to their complexity.

Dependencies on third-party software components and the impact of software defects have been studied extensively [13, 26, 29, 39]. However, these problems account for only a part of the change-management faults. Distributed

[1] According to the survey, the average upgrade-failure rate was 8.6%, with some administrators reporting that up to 50% of upgrades had failed in their respective installations.

dependencies and the impact of operator errors are not well understood. Several classifications of change-management faults have been proposed [5, 10, 13, 17, 23, 25], but, in many cases, the fault categories are not disjoint, the underlying criteria for establishing these categories remain unclear, or the classifications are relevant only for subsets of the change-management faults.

In this paper, we establish a rigorous taxonomy of change-management faults. We analyze fault data from three sources, collected independently using different methodologies: a user study, a survey and a field study. Using statistical cluster analysis, we emphasize the similarities between some of these faults and we show that they form four distinct groups: simple configuration errors (*e.g.* typos), semantic configuration errors, shared-library problems and complex procedural errors.

1.1 Related Work

Software defects, also known as software bugs, have been studied extensively, and rigorous taxonomies have been proposed [31, 39]. However, Crameri *et al.* [34] remark that most upgrade failures are not due to software defects, but to faults in the upgrade procedure, such as broken dependencies.

Anderson [2] describes three mechanisms that are commonly responsible for breaking dependencies on dynamically-linked libraries (DLLs) in the Windows NT operating system — a phenomenon colloquially known as "DLL Hell". More recently, Dig *et al.* [13] examine how the natural evolution of APIs can impact upgrades by breaking dependencies on third-party Java frameworks and libraries. They conclude that 81% – 100% of the breaking API changes are refactorings (reorganizations of the code structure) and less than 30% are intended behavioral changes (modified application semantics).

API compatibility is not the only class of dependency that can be broken during an upgrade. Oppenheimer *et al.* [10] study 100+ *post-mortem* reports of user-visible failures from three large-scale Internet services. They classify failures by location (front-end, back-end and network) and by the root cause of the failure (operator error, software fault, hardware fault). Most failures reported occurred during change-management tasks, such as scaling or replacing nodes, and deploying or upgrading software.

Nagaraja *et al.* [23] conduct a user study with 21 operators who perform three change-management and three firefighting tasks on a small e-commerce system called RUBiS [7]. The authors observe seven classes of faults, ordered by frequency: global misconfiguration, local misconfiguration, start of wrong software version, unnecessary restart of software component, incorrect restart, unnecessary hardware replacement, wrong choice of hardware component.

Oliveira *et al.* [17] present a survey of 51 database administrators, who report eight classes of faults, ordered by frequency: deployment problems, performance problems, general-structure problems, DBMS problems, access-privilege problems, space problems, general-maintenance problems, and hardware problems. While the database administrators from the survey spend only 46% of their time performing change-management tasks, all the faults reported can occur during change management.

Reason [36] presents the Generic Error-Modeling System (GEMS), which identifies three cognitive levels at which humans solve problems and make mistakes: the skill-based level (responsible for 60% of errors); the rule-based level (responsible for 30% of errors); and the knowledge-based level (responsible 10% of errors). Building on this framework, Keller *et al.* [25] propose a model of configuration errors, with three categories: typographical errors (corresponding to the skill-based level); structural errors (which may occur on any cognitive levels); and semantic errors (corresponding to the knowledge-based level).

Brown *et al.* [5] propose a model for quantifying the configuration complexity from a human perspective, taking into account the "context switches" between different mental models for the configuration process. The model quantifies the configuration complexity according to three dimensions: the execution complexity (the actions that an administrator must perform); the parameter complexity (the amount of parameters that the administrator must understand and tune); and the memory complexity (related to parameters that must be supplied in more than one configuration

action). The authors propose several metrics for each of these three dimensions. Lin *et al.* [30] continue this research by studying the complexity involved in the decision-making process required for system administration.

These models do not constitute a rigorous taxonomy of change-management faults. Some classifications are too coarse-grained (*e.g.*, the fault location [10]) and do not provide sufficient information about the fault. Other classifications (*e.g.*, the breaking API changes [13] and the typographical/structural/semantic configuration errors [25]) are relevant for only a subset of the change-management faults (broken dependencies on third-party frameworks and components and configuration actions that require file-editing, respectively). In many cases, the fault categories are not disjoint and the criteria for establishing these categories are not clearly stated. Most of the configuration-complexity metrics [5] characterize the entire configuration procedure, rather than an individual action that might fail, and it remains unclear whether a higher configuration complexity leads to a higher incidence of change-management faults. In this paper, we annotate fault data from three different sources with categories from some of these existing models, and we use statistical cluster analysis to establish a taxonomy of change-management faults.

2 Change-management fault data

We collect data from three sources: a 2004 *user study* of system-administration tasks in an e-commerce system [23], a 2006 *survey* of database administrators [17] and a previously unpublished *field study* of bug reports filed in 2007 for the Apache web server [1]. We categorize these faults according to the models proposed in [10, 17, 23, 25, 36], which are reviewed in Table 1. When describing and analyzing this data, we use a uniform terminology:

- *Task*: The actions taken by an administrator in order to achieve a pre-defined goal. System-administration tasks are traditionally classified in two categories [10]: regular-maintenance (change-management and other tasks performed in the normal operating mode of the system) and firefighting (tasks performed in order to recover from failures). A task is accomplished through a *procedure*, which is composed of one or several *actions* [5].

- *Fault*: The root cause of a change-management failure. Faults that occur during change-management tasks include hardware faults, software defects or operator errors [10]. Some faults, *e.g.*, hardware faults, are independent of the task performed; however, by reducing the system capacity or by rendering certain resources unavailable for extended periods of time, change-management tasks introduce a window of vulnerability when a random dependency-fault might have a critical, adverse effect.

- *Fault Impact*: The measurable outcome of a fault. Some faults, *e.g.*, disabling an application server in the middle tier, might be masked by the system redundancy. Other faults might generate failures such as increased latency, throughput degradation, complete system outage, data loss or latent errors.

User study. Nagaraja *et al.* [23] conduct a user study with 21 system administrators, with varying degrees of experience, who perform maintenance and firefighting on RUBiS (the Rice University Bidding System) [7], an open-source online bidding system, modeled after eBay. The system administrators are given three change-management tasks — adding an application server, upgrading the database machine, and upgrading one web server — and three diagnose-and-repair tasks. The study reports 32 instances of 16 unique faults, and 10 misdiagnoses; 25 instances of 13 unique faults occur during the change-management tasks. The study reports the fault locations, their root causes, and it classifies the faults into seven categories (see Table 1), which constitute a refinement of the root-cause classification. From the detailed descriptions provided, we infer the cognitive level where these faults occur and, where applicable, the configuration-error subcategory. The most frequent failures are global misconfigurations, which compromise the communication between components from different tiers of the system and. These errors occur on the skill-based and

Table 1: Fault categories.

Variable	Categories	Description
Location [10]	front-end	Fault in the front end of the infrastructure, which handles the client connections
	middle tier	Fault in the middle tier of the infrastructure, where the application servers process client requests
	back-end	Fault in the backend of the infrastructure, where persistent data is stored (typically, in a database)
Root cause [10]	software	Software defect
	hardware	Failure of a hardware component
	configuration	Operator error while configuring the system
	procedure	Operator error during the task procedure (*e.g.*, stopping/starting a server)
Fault classification:	global misconfiguration	Inconsistencies in one or more configuration files compromising the communication between system components
	local misconfiguration	Configuration error affecting a single node in the system
	start of wrong SW version	Configuring one version and starting a different version of a software component
from user study [23]	unnecessary restart	Unnecessarily restarting a software component (*e.g.*, the database)
	incorrect restart	Starting a software component incorrectly (*e.g.*, without obtaining the necessary access privileges)
	unnecessary HW replacement	Misdiagnosing the service malfunction as a hardware problem
	wrong choice of hardware	Installing the database on a slow disk
	deployment	Changes to the online system (previously tested offline) cause the database to misbehave
	performance	The DBMS delivers poor performance to the application or user
	general structure	Incorrect database design or unsuitable changes to the database schema
from survey [17]	DBMS	Software defects in the DBMS
	access privileges	Insufficient/excessive access-privileges granted to users or applications
	space	Disk space or tablespace exhaustion
	general maintenance	Other problems (*e.g.* incompatible upgrades, incorrect restarts)
	hardware	Hardware failure and (potential) data loss
	build	Wrong compile flags or paths, missing or conflicting libraries prevent compilation
	paths and permissions	Incorrect paths to files or insufficient access permissions
	environmental conflicts	Wrong library versions, byte orders, file separators
from field study [1]	third-party error	Bugs and misconfigurations in third-party components
	parameter tuning	Incorrect setting for categorical, integer or real-valued parameter
	other error	Other errors in the application's configuration file (*e.g.*, missing commands, wrong order of commands, use of wrong command, typos, syntax errors)
Cognitive level [36]	skill-based	Slips and lapses, occurring during the common, repetitive tasks
	rule-based	Mistakes when reasoning and solving problems through pattern-matching
	knowledge-based	Mistakes when reasoning from first principles
Configuration files [25]	typographical	Spelling errors (*i.e.*, typos)
	structural	Configuration directives misplaced or similar, but incorrect, format used
	semantic	Constraints among parameters ignored or unknown parameters used

rule-based cognitive levels, due to a mental "context switch" [5] between configuration actions performed on different components.

Survey. Oliveira *et al.* [17] conduct a survey of 51 database administrators (DBAs), with between 2 and 10+ years of experience. The DBAs report that their regular-maintenance tasks are related to change management (*e.g.*, tuning the performance, changing the database structure, modifying the data, coding and upgrading the software), to runtime monitoring (*e.g.*, space monitoring/management, system monitoring, integrity checks, performance monitoring), and to recovery preparations (*e.g.*, making/testing backups, conducting recovery drills). The survey identifies eight classes of faults (see Table 1) and provides details for 20 unique, individual faults. All these faults occur in the system back-end. From the descriptions provided, we infer the root cause of these faults, the cognitive level where they and, where applicable, the configuration-error subcategory.

Field study. We analyze the reports filed in the bug database of the Apache web server (v. 2.2) [1] between 1 January 2007 and 21 December 2007. We focus on closed bugs that have been marked `FIXED` – which correspond to software defects –, `INVALID` or `WONTFIX` – which indicate operator errors. These operator errors have serious impacts, which prompted the opening of a bug report, but they have been excluded from previous studies mining the Apache bug database (*e.g.* [26, 29]), which focus on classifying software defects. Because bug reports are terse, the tasks performed when these faults occurred remain unclear; however, after we filter out presentation bugs (*e.g.*, documentation, maintainability issues), the remaining faults are likely to be related to change-management tasks. To provide a starting classification for these faults, as for the user study and the survey, we bin them in six categories: build, paths and permissions, environmental conflicts, third-party error, parameter tuning, other error (see Table 1). We try to infer the location of the fault from the bug description. If the bug report was filed for a component that provides functionality typical for front-ends (*e.g.*, `mod_proxy`, `mod_rewrite`), we consider that the fault occurred on the front-end. If the component is used for authentication or computationally-intensive operations (*e.g.*, LDAP, PHP), we consider that the fault occurred on the middle tier.

It is interesting to note that some bugs are reported in several sources. For instance, a configuration error where Apache is instructed to serve static HTML files from an existing, but incorrect, location is reported in both the user and field studies. A configuration error, where the application is granted insufficient access privileges to database tables, is reported in both the user study and the survey. A procedural error, where the wrong version of the Apache server is started in the front-end, is reported during three different tasks of the user study, which were performed by different operators. Another procedural error, where the application queries an incorrect database schema, is reported in conjunction with four different tasks in the survey. A configuration error, which prevents Apache from sending files larger than 64K, was the source of two bug reports in the field study. To keep track of these duplicated fault reports, we assign each unique fault a name and we annotate all the fault's instances with this name. Faults that occur in the same way (*e.g.*, configuring the wrong port for a server) but are located on different tiers are considered different faults, because they are likely to be introduced by different kinds of operators (*e.g.*, system administrators, database administrators, application developers) who use different mental models for the tasks they perform.

For consistency, we consider that all path-related faults occur on the rule-based cognitive level, and that all the database-schema faults occur on the knowledge-based level. We consider that faults related to file-system and database access-privileges are configuration errors, even if they don't involve editing configuration files, and they occur on the rule-based cognitive level. The annotated faults used in the classification are listed in Appendix A, and they can also be downloaded from `http://www.ece.cmu.edu/~tdumitra/upgrade_faults/`.

3 Model of change-management faults

Our goal is to establish a taxonomy of upgrade faults. We annotate each fault reported with several *classification variables*, which correspond to the existing fault models. A *numerical* variable (*e.g.*, whether the configuration complexity of the task that has produced the fault) indicates the magnitude of the variable's instance for a particular fault. A *categorical* variable (*e.g.*, the fault occurred in the frontend, the middle tier or the backend) shows if a fault belongs to one of several categories. A *binary* variable (*e.g.*, whether the fault was independent of the task) shows a trait that is either present or absent in a fault (note that, when both levels of a two-level variable are to be treated on a par, the variable is categorical rather than binary).

We use three principles for selecting the relevant faults and classification-variables from the data described in Section 2.

P1. Software defects are orthogonal from upgrading concerns.

While some upgrades fail due to software defects [34] these defects occur for reasons that are not related to the upgrade, and they might be exposed in other situations as well. Similarly, hardware defects are not always related to an upgrade. We therefore exclude software and hardware defects from our fault model; for instance, in the field study, we consider only bug reports marked as `INVALID` or `WONTFIX`.

P3. We classify upgrade faults, not fault impacts.

The fault impact depends not only on the upgrade fault, but also on the system architecture. For instance, if the system emplys replicated application servers in the middle tier, a fault that disables one server may be masked by the redundancy in the architecture. We therefore exclude the fault impact from the classification variables, to avoid establishing a connection among distinct faults, which occur in different ways, but which have similar impacts on the system.

P4. We classify upgrade faults, not upgrade tasks.

Similarly, an upgrade fault can occur during different upgrade tasks. To avoid establishing a connection among multiple faults for the sole reason that they were recorded during the same task, we exclude the task and the configuration-complexity metrics [5] from the classification variables.

All the classification variables we use are categorical, and they are described in Table 1. To avoid placing identical faults in different categories, we merge their original fault classifications, originating from different studies. This pre-processing step merges the "access-privilege problems" from the survey with the "global misconfigurations" from the user study, and the "path and permissions" classification from the field study with the "local misconfigurations" from the user study.

3.1 Methodology

We compare the faults using *Gower's similarity coefficient* [19], which is widely used in the natural sciences for establishing taxonomies of living organisms. This is a generalized measure for how closely related two samples are, and it can be computed for binary, categorical or numerical data. For two samples i and j, characterized by n variables, Gower's coefficient G_{ij} is the sum of the similarity scores for all the variables normalized by the number of comparable variables:

$$G_{ij} = \frac{\sum_{k=1}^{n} s_{ijk}}{\sum_{k=1}^{n} \delta_{ijk}}, \qquad \text{where } s_{ijk} \in [0,1] \text{ and } \delta_{ijk} \in \{0,1\}$$

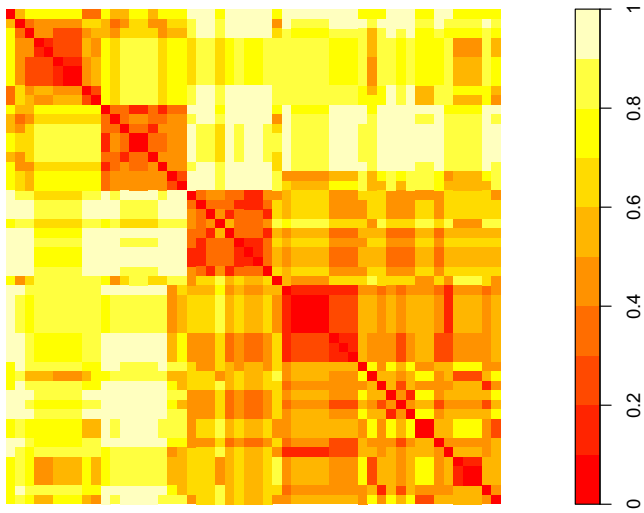

Figure 1: Heat map of the Gower distance for the faults described in Section 2. The lighter colors indicate higher distances; the elements on the diagonal are 0.

δ_{ijk} is 1 when i and j are comparable and 0 when the comparison is impossible, because information is missing or in the case of binary variables indicating the absence of a trait from both samples. The score s_{ijk} is 0 when i and j are considered different and a positive fraction or 1 when they have some degree of similarity, defined as follows:

$$s_{ijk} = \begin{cases} 1 & \text{if both } i \text{ and } j \text{ have the trait indicated by variable } k \\ 0 & \text{otherwise} \\ 1 & \text{if } i \text{ and } j \text{ belong to the same category of variable } k \\ 0 & \text{otherwise} \\ 1 - \frac{|x_i - x_j|}{R_k} & \text{where } R_k \text{ is the range of variable } k \end{cases} \quad \begin{matrix} \left.\begin{matrix} \\ \\ \\ \\ \end{matrix}\right\} \\ \\ \end{matrix} \begin{matrix} \text{for binary data} \\ \text{for categorical data} \\ \text{for numerical data} \end{matrix}$$

When the two samples belong to the same category of a two-level categorical variable k, the samples are considered comparable and similar with respect to variable k ($\delta_{ijk} = 1$ and $s_{ijk} = 1$). Contrariwise, when both samples lack the trait indicated by a binary variable k, the samples cannot be compared ($\delta_{ijk} = 0$ and $s_{ijk} = 0$).

Gower's coefficient is 1 if the samples i and j are identical and 0 if they belong to different categories (for categorical variables) and are located at opposite ends of the range (for numerical variables); otherwise the coefficient is a positive fraction ($G_{ij} \in [0, 1]$). If none of the n variables are comparable for samples i and j, then G_{ij} is undefined. For

categorical data without any missing information, G_{ij} is the fraction of variables that have matching values in the two samples.

We cluster the faults described in Section 2 by computing the *Gower distance* ($d_G(i,j) = \sqrt{1 - G_{ij}}$) between each pair of faults, as shown in Figure 1. Because the Gower distance satisfies the triangle inequality, it can substitute the Euclidian distance in statistical cluster-analysis techniques.

We start by placing each fault in a separate cluster, and then we merge clusters through an iterative algorithm. At each step, we merge two clusters with the goal of minimizing the distance variance within clusters. The cluster variance is defined as the sum of the squares of the distance between all objects in the cluster and the centroid of the cluster; for instance, the variance of a cluster that contains only identical faults is 0. The result of the algorithm is a binary tree where the leaves correspond to the individual faults and the root corresponds to a cluster encompassing all the faults. This approach, known as *Ward's hierarchical clustering method* [24], emphasizes the natural, compact clusters in the data by showing how the within-cluster variance increases at each level in the tree, as shown in Figure 2. To validate our results, we also perform k-means clustering [24] on the fault data. Instead of creating a hierarchy of clusters, this approach attempts to partition the samples into disjoint subsets and indicates whether the clusters are overlapping.

3.2 Interpretation

The *dendrogram* from Figure 2 shows that our fault data contains four natural clusters:

Type 1, on the right, contains simple faults (typos or structural) that occur when editing configuration files. Most modern enterprise systems check the syntax of their configuration files at startup in an attempt to prevent this class of faults [25].

Type 2, on the middle-right, corresponds to complex configuration errors, which occur on the knowledge-based cognitive level and which indicate a misunderstanding of the configuration directives used. Currently, we lack any automated techniques for handling these faults.

Type 3, on the left, contains simple faults due to to missing libraries or port conflicts, which occur at compile-time or at run-time. These can be either procedural faults or configuration faults that do not occur while modifying configuration files. "DLL Hell" faults [2] fall in this category.

Type 4, on the middle-left, includes complex faults that do not occur while modifying configuration files. These faults can occur on any cognitive level and in any tier of the system, and they are sometimes independent of the task performed. These faults typically affect the communication between distributed components by preventing access to certain services or database tables or by degrading the performance. Incorrect or unnecessary restarts are a distinct subset of this cluster. To the best of our knowledge, these faults have not been characterized as a distinct group before, despite their strong similarity suggested by Figure 2.

Types 1 and 3 form a larger cluster, which corresponds to faults that occur while editing configuration files. Types 2 and 4 form another large cluster, which corresponds to the other procedural and configuration faults. The clear separation between these two high-level clusters corresponds to the intuition that editing configuration files and performing actions through the command line or through a GUI require different mental models and lead to different types of faults.

The *cophenetic correlation coefficient* for the dendrogram from Figure 2, which shows the correlation between the Gower distance and the distance in the cluster tree, is 0.78. The k-means clustering technique produces almost the same four clusters, with three exceptions: fault `mysql_nopass_root`, which is placed in cluster 1, and faults `no_space`

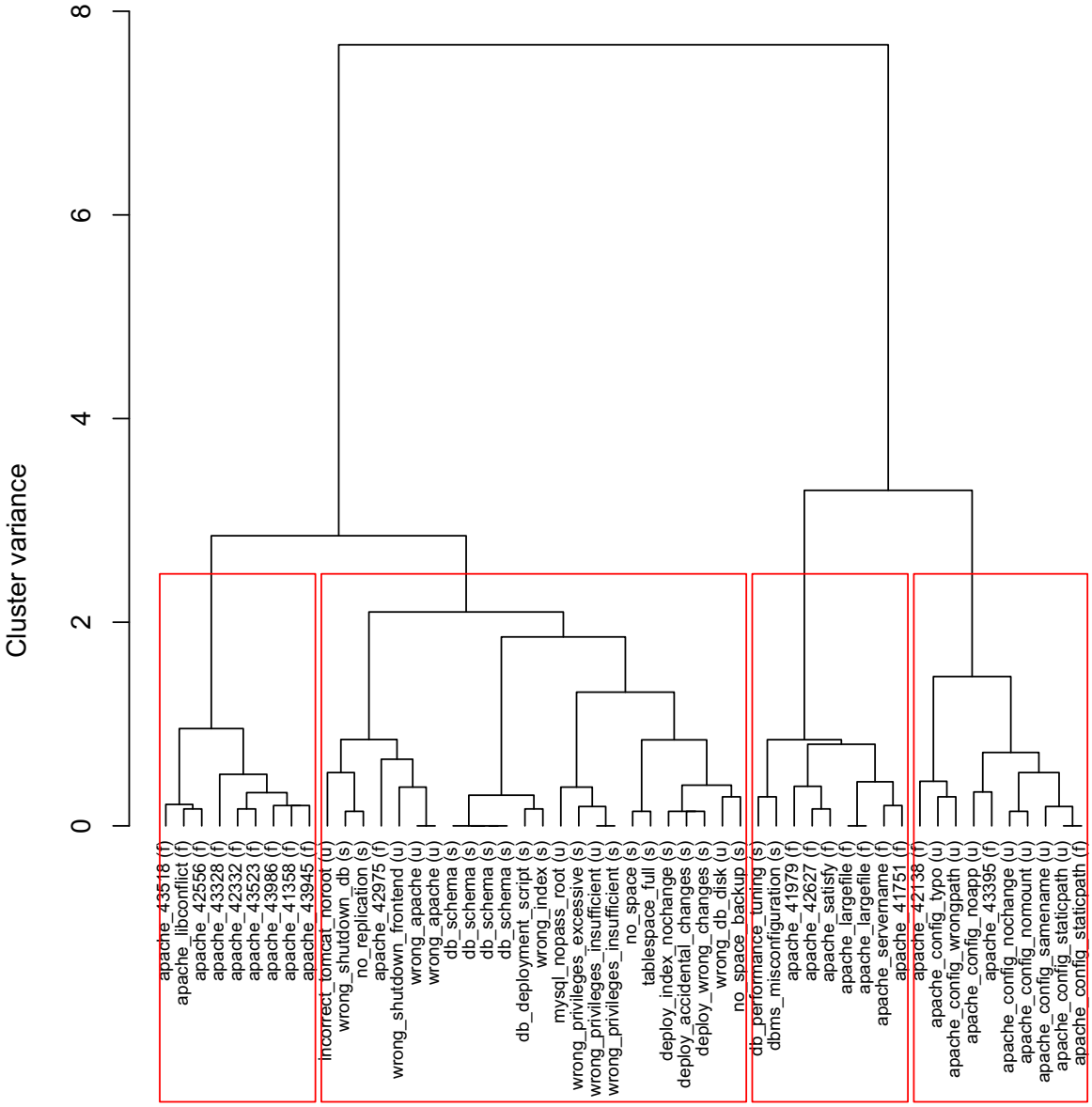

Figure 2: Fault-model dendrogram, constructed using Ward's hierarchical clustering method. The leaves correspond to the faults reported in the user study (u), the survey (s) and the field study (f). A horizontal line indicates two clusters that are linked together into a larger cluster. The position of these lines on the y-axis indicates the variance within the cluster. A link variance that is significantly larger than the variance of the links below suggests the presence of natural clusters in the data (highlighted by the red rectangles in the figure).

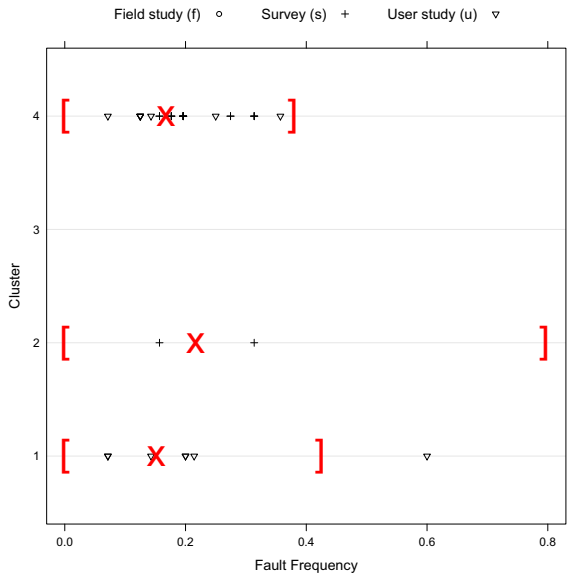

Figure 3: Frequencies of the four fault types.

and `tablespace_full`, which are placed in cluster 4. This suggests that there is some overlapping between these clusters.

3.3 Frequencies of fault types

We also estimate how frequently these fault types occur during an upgrade, by considering the percentage of operators who induced the fault (during the user study) or the percentage of DBAs who consider the specific fault among the three most frequent problems that they have to address in their respective organizations (in the survey). We cannot derive frequency information from the field-study data. The individual estimations are imprecise,[2] because the rate of upgrades is likely to vary among organizations and administrators, and because of the small sample sizes (5–51 subjects) used in these studies. We therefore estimate the fault frequency by combining the individual estimations of for each fault type. This technique computes the maximum-likelihood estimation by minimizing the sum of squared errors from the dissimilar estimates [8].

Using this methodology, we estimate that Type 1 faults occur in 15.2 % of upgrades, with a 95% confidence interval of $[0, 42.3]$, and that Type 4 faults occur in 16.8 % of upgrades, with a 95% confidence interval of $[0, 37.8]$. Types 2 and 3 were predominantly reported in the field-study, so we lack sufficient information to compute statistically-significant fault frequency for these clusters.

[2]The *precision* of a measurement indicates if the results are repeatable, with small variations, and the *accuracy* indicates if the measurement is free of bias. While in general it is not possible to improve the accuracy of the estimation without knowing the systematic bias introduced in an experiment, combining multiple measurements can improve the precision of the estimation [8].

3.4 Threats to validity

The data collected from the three sources has certain characteristics that might skew the results of the cluster analysis. Because the user study is concerned with the behavior of the operators, it does not report any software defects or hardware failures. Faults that are independent of the task performed are typically reported in surveys, but not in user studies, which evaluate the outcome of a set of tasks. Moreover, in our field study the tasks were unknown, so we are unable to determine whether the fault is task-dependent or not. We have only two independent faults in our data, `no_space` and `tablespace_full`. Hierarchical clustering places them in cluster 2 and k-means clustering places them in cluster 4, which suggests that task-dependence might have a higher impact on fault classification than suggested in our results. Configuration errors submitted as bugs tend to be due to significant misunderstandings of the program semantics, and, as a result, our field study contains an unusually-high number of faults occurring on the knowledge cognitive level. Finally, the results of bug search are not repeatable because the status of bugs changes over time; in particular, more open bugs are likely to be marked as invalid or not fixed in the future.

4 Tolerating upgrade faults

Modern enterprise software-systems check the syntax of their configuration files, and they are able to detect 38%–83% of the Type 1 faults at startup [25]. Type 2 faults are harder to detect automatically. Keller *et al.* argue that checking the constraints among parameter values can improve the robustness against Type 2 faults [25], while Zheng *et al.* show how to generate configurations that tune a specific metric (*e.g.*, server-side throughput) by solving a constrained-optimization problem.

To prevent Type 3 faults, modern operating systems provide package managers that determine automatically how to install a new package, or upgrade an existing one, along with all of its dependencies. These tools include APT [37] for Debian Linux, YUM [6] for RedHat Linux, Portage [42] for Gentoo Linux and the Windows Update Agent [32] for Microsoft Windows. Package-management systems maintain repositories of packaged software components, along with metadata that tracks the dependency and conflict relationships among all the packages in the repository. For example, Figure 4 shows the dependencies, obtained from the APT package manager, among the software components from the middle-tier node of a typical RUBiS infrastructure (such as the one used in the user study described in Section 2). The middle-tier node includes Apache web server with the PHP interpreter and the MySQL client library, which depend on a complex graph of third-party components. Because Figure 4 does not capture the distributed dependencies among the application components and because industry-standard e-commerce applications typically require additional services, such as directory servers, security infrastructures and backup mechanisms [18], this graph represents but a small fraction of the complex dependency relationships among the components of a real-world distributed-system.

In practice, dependencies in distributed systems are often poorly documented [13, 15], and they cannot always be detected automatically. Dependency-mining techniques, such as static analysis [12, 40], semantic analysis [12], runtime monitoring [14] or active perturbation [4], cannot provide a complete coverage of all the factors that might influence the behavior of a distributed system. For example, the complete set of shared libraries that might be loaded by an application cannot be discovered using static analysis (*e.g.*, if the libraries are specified external to the code, through configuration settings or environment variables), and monitoring the library-loading operations at runtime is a best-effort approach that might not cover all of the possible application-behaviors. Dependency tracking ultimately relies on metadata that is partially maintained, manually, by teams of developers and quality-assurance engineers through a time-intensive and error-prone process.

Moreover, determining the correct configuration of an upgraded system by resolving the dependencies of the installed components is an NP-complete problem [11]. Current approaches use heuristics [6, 37] to ensure that the search for a correct configuration terminates in a timely fashion, or they rely on SAT solvers [11, 41] to guarantee that

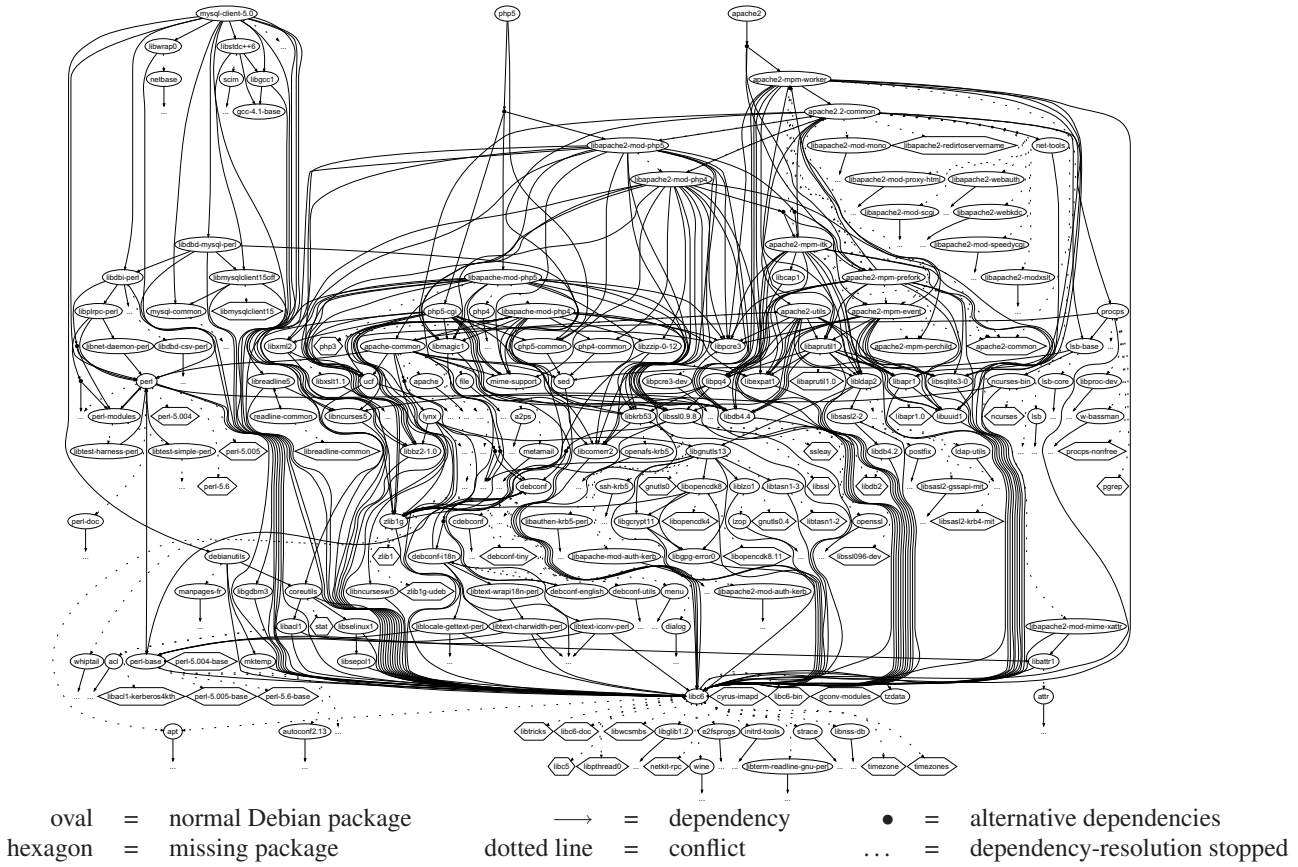

Figure 4: Package dependencies on a middle tier node.

a solution will be found for all installable packages. Heuristics-based approaches might fail to find a solution where one exists, while, for certain corner cases, the run-time of SAT solvers is exponential in the size of the repository [41]. These approaches might be adequate for the current sizes and structures of software repositories [16, 41], but they are likely to fail as the amount of dependency-related metadata increases. For instance, current repositories contain 10,000 – 25,000 packages and 70,000 – 85,000 inter-package dependencies [28]. If we include configuration dependencies – needed to prevent Type 1 and 2 faults – the size of these repositories would increase by one order of magnitude (*e.g.*, a typical instance of the Windows registry has approximately 200,000 configuration settings that may be shared by several applications [43]). Best practices in distributed-system administration recommend the use of a Configuration-Management Database (CMDB) that centralizes all of the dependency information in the system [35]; the size of a complete CMDB would likely expose the fundamental limitations of the current online-upgrade approaches in terms of their need to track dependencies.

The current effect of these fundamental limitations is that dependencies can sometimes remain *hidden*, because they cannot be detected automatically or because the are overlooked due to their complexity, which can lead to upgrade failures due to broken dependencies. Furthermore, we currently lack automated techniques for handling Type 4

faults. Oliveira *et al.* propose checking the actions of database administrators, using real workloads in a "validation environment," but they also remark that such a validation is difficult when the administrator's goal is to change the database schema or the system's observable behavior. As enterprise systems grow larger and more complex, novel approaches for understanding and tolerating upgrade faults will be needed in order to prevent hidden dependencies from sustaining a steady increase in upgrade-failure rates.

5 Conclusion

We propose a novel fault-model for upgrades in distributed systems, with four categories. Three of these categories correspond to the previously-known simple configuration errors (*e.g.*, typos), semantic configuration errors, and broken environmental dependencies. We identify a new type of fault – complex procedural errors –, which can disrupt the communication between distributed components in a modern enterprise system by breaking dependencies on remote components or on data objects. We review the existing approaches for tolerating these faults, and we show that their fundamental limitations prevent them from improving the reliability of upgrades in distributed systems.

References

[1] Apache Bugzilla. `https://issues.apache.org/ bugzilla/`. Retrieved on 21 Dec 2007.

[2] R. Anderson. The end of DLL Hell. *MSDN Magazine*, Jan 2000.

[3] S. Beattie, S. Arnold, C. Cowan, P. Wagle, and C. Wright. Timing the application of security patches for optimal uptime. In *Large Installation System Administration Conference*, pages 233–242, Philadelphia, PA, Nov 2002.

[4] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Integrated Network Management*, pages 377–390, Seattle, WA, May 2001.

[5] A. Brown, A. Keller, and J. L. Hellerstein. A model of configuration complexity and its application to a change management system. In *IFIP/IEEE Symposium on Integrated Network Management*, pages 631–644, Nice, FR, May 2005.

[6] R. Brown and J. Pickard. *Yum (Yellowdog Updater, Modified) HOWTO*, Sep 2003. `www.phy.duke.edu/~rgb/General/yum_HOWTO/yum_HOWTO`.

[7] C. Amza *et al.* Specification and implementation of dynamic web site benchmarks. In *IEEE Workshop on Workload Characterization*, pages 3–13, Austin, TX, Nov 2002. `http://rubis.objectweb.org/`.

[8] C. Chatfield. *Statistics for Technology: A Course in Applied Statistics*. Chapman & Hall/CRC, 3rd edition, 1983.

[9] China Tech News. Symantec-Norton software upgrade failure causes computer collapse. http://www.chinatechnews.com/2007/05/21/5416-symantecs-norton-software-upgrade-failure-causes-computer-collapse/, May 2007.

[10] D. Oppenheimer et al. Why do Internet services fail, and what can be done about it? In *USENIX Symposium on Internet Technologies and Systems*, Seattle, WA, Mar 2003.

[11] R. Di Cosmo. Report on formal management of software dependencies. Technical report, INRIA, Sep 2005. (EDOS Project Deliverable WP2-D2.1).

[12] D. Dig, C. Comertoglu, D. Marinov, and R. Johnson. Automated detection of refactorings in evolving components. In *European Conference on Object-Oriented Programming*, pages 404–428, Nantes, France, Jul 2006.

[13] D. Dig and R. Johnson. How do APIs evolve? A story of refactoring. *Journal of Software Maintenance and Evolution (JSME)*, 18(2):83–107, Mar/Apr 2006.

[14] 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 *International Conference on Autonomic Computing*, pages 106–113, New York, NY, May 2004.

[15] A. Egyed. A scenario-driven approach to trace dependency analysis. *IEEE Transactions on Software Engineering*, 29(2):116–132, 2003.

[16] F. Mancinelli et al. Managing the complexity of large free and open source package-based software distributions. In *International Conference on Automated Software Engineering*, pages 199–208, Tokyo, Japan, Sep 2006.

[17] F. Oliveira et al. Understanding and validating database system administration. *USENIX Annual Technical Conference*, Jun 2006.

[18] M. Galic, A. Halliday, A. Hatzikyriacos, M. Munaro, S. Parepalli, and D. Yang. *A Secure Portal using WebSphere Portal V5 and Tivoli Access Manager V4.1*. IBM Redbooks, Dec 2003.

[19] J. C. Gower. A general coefficient of similarity and some of its properties. *Biometrics*, 27(4):857–871, Dec 1971.

[20] J. Gray. Why do computers stop and what can be done about it? In *Symposium on Reliability in Distributed Software and Database Systems*, pages 3–12, Los Angeles, CA, 1986.

[21] J. Gray. A census of Tandem system availability between 1985 and 1990. *IEEE Transactions on Reliability*, 39(4):409–418, Oct 1990.

[22] J. Hart and J. D'Amelia. An analysis of RPM validation drift. In *USENIX Large Installation System Administration Conference*, pages 155–166, Philadelphia, PA, Nov 2002.

[23] K. Nagaraja et al. Understanding and dealing with operator mistakes in Internet services. In *USENIX Symposium on Operating Systems Design and Implementation*, pages 61–76, San Francisco, CA, Dec 2004.

[24] L. Kaufman and P. J. Rousseeuw. *Finding Groups in Data: an Introduction to Cluster Analysis*. Wiley Series in Probability and Mathematical Statistics. Wiley, 1990.

[25] L. Keller, P. Upadhyaya, and G. Candea. ConfErr: A tool for assessing resilience to human configuration errors. In *International Conference on Dependable Systems and Networks*, Anchorage, AK, Jun 2008.

[26] S. Kim, T. Zimmermann, J. E. James Whitehead, and A. Zeller. Predicting faults from cached history. In *International Conference on Software Engineering*, pages 489–498, Minneapolis, MN, May 2007.

[27] C. Koch. AT&T Wireless self-destructs. *CIO Magazine*, Apr 2004. `www.cio.com/archive/041504/wireless.html`.

[28] N. LaBelle and E. Wallingford. Inter-package dependency networks in open-source software. In *International Conference on Complex Systems*, Boston, MA, Jun 2006.

[29] Z. Li, L. Tan, X. Wang, S. Lu, Y. Zhou, and C. Zhai. Have things changed now?: an empirical study of bug characteristics in modern open source software. In *ASPLOS Workshop on Architectural and System Support for Improving Software Dependability*, pages 25–33, 2006.

[30] B. Lin, A. B. Brown, and J. L. Hellerstein. Towards an understanding of decision complexity in IT configuration. In *Symposium on Computer-Human Interaction for the Management of Information Technology*, 2007.

[31] S. Lu, Z. Li, F. Qin, L. Tan, P. Zhou, and Y. Zhou. BugBench: A benchmark for evaluating bug detection tools. In *PLDI Workshop on the Evaluation of Software Defect Detection Tools*, Chicago, IL, Jun 2005.

[32] Microsoft Developer Network. *Windows Update Agent*. `http://msdn2.microsoft.com/en-us/library/aa387099.aspx`. Retrieved on 18 Feb 2008.

[33] Neumann, P. *et al.* America Offline. *The Risks Digest*, 18(30–31), Aug 8–9 1996. `http://catless.ncl.ac.uk/Risks/18.30.html`.

[34] O. Crameri et al. Staged deployment in Mirage, an integrated software upgrade testing and distribution system. In *Symposium on Operating Systems Principles*, pages 221–236, Stevenson, WA, Oct 2007.

[35] Office of Government Commerce. Information technology infrastructure library (ITIL), 2001.

[36] J. Reason. *Human Error*. Cambridge University Press, 1990.

[37] G. N. Silva. *APT HOWTO*, Aug 2005. `www.debian.org/doc/manuals/apt-howto/index.en.html`.

[38] Skype. What happened on Aug 16. `http://share.skype.com/sites/en/2007/08/what_happened_on_august_16.html`, Aug 2007.

[39] M. Sullivan and R. Chillarege. Software defects and their impact on system availability-a study of field failures in operating systems. In *Fault-Tolerant Computing Symposium*, pages 2–9, 1991.

[40] Y. Sun and A. Couch. Global impact analysis of dynamic library dependencies. In *USENIX Large Installation System Administration Conference*, pages 145–150, San Diego, California, Dec 2001.

[41] C. Tucker, D. Shuffelton, R. Jhala, and S. Lerner. OPIUM: Optimal package install/uninstall manager. In *International Conference on Software Engineering*, pages 178–188, Minneapolis, MN, May 2007.

[42] S. Vermeulen, G. Goodyear, R. Marples, D. Robbins, C. Houser, and J. Alexandratos. *Gentoo Handbook*, May 2007. `http://www.gentoo.org/doc/en/handbook/handbook-x86.xml`.

[43] Wang, Y.-M. *et al.* STRIDER: A black-box, state-based approach to change and configuration management and support. In *USENIX Large Installation System Administration Conference*, pages 159–172, San Diego, CA, Oct 2003.

Appendix A Fault Data

Fault name	Unique name	Source	Page/Bug Id	Description	Location	Root cause	Original classification	Config. file fault	Cognitive level	Reported effect	Cluster
apache_config_noapp	apache_config_noapp	[nagaraja04]	4	Added info about the new app server; but forgot to add the machine's name to the line that specifies app-server names	frontend	Configuration	Local misconfiguration	structural	knowledge	Latent error (frontend cannot contact app server)	1
apache_config_typo	apache_config_typo	[nagaraja04]	4	Syntax error in Apache configuration file	frontend	Configuration	Local misconfiguration	typo	skill	Throughput degradation (mod_jk crashed)	1
apache_config_samename	apache_config_samename	[nagaraja04]	4	Configured identical names for the application servers in the Tomcat connector	frontend	Configuration	Local misconfiguration	structural	rule	Service inaccessible (mod_jk crashed on both frontend servers)	1
apache_config_nochange	apache_config_nochange	[nagaraja04]	5	Forgot to modify the Apache configuration file altogether	frontend	Configuration	Global misconfiguration	structural	rule	Throughput degradation	1
apache_config_nomount	apache_config_nomount	[nagaraja04]	6	Forgot to specify how Apache should map a given URL to a request for a Tomcat servlet	frontend	Configuration	Global misconfiguration	structural	rule	Throughput degradation (affected web server unable to process client requests)	1
apache_config_oldpath	apache_config_staticpath	[nagaraja04]	6	Configured new server to get static files / heartbeat program from the old Apache's directory tree	frontend	Configuration	Local misconfiguration	structural	rule	Latent error (when old distribution removed; new server cannot serve static files)	1
apache_config_wrongpath	apache_config_wrongpath	[nagaraja04]	6	Incorrectly specified the path to the heartbeat program	frontend	Configuration	Global misconfiguration	typo	skill	Throughput degradation (mod_jk crashed)	1
apache_42138	apache_42138	Apache Bugzilla	42138	Httpd.conf Error		Configuration	Other error	typo	skill	Error message	1
apache_43111	apache_config_staticpath	Apache Bugzilla	43111	/htdocs getting served when not configured	frontend	Configuration	Local misconfiguration	structural	rule	Error message	1
apache_43395	apache_43395	Apache Bugzilla	43395	RewriteRule and Location or Directory does not work	frontend	Configuration	Other error	structural	knowledge	Error message	1
wrong_apache1	wrong_apache	[nagaraja04]	5	Reconfigured one Apache distribution and launched the executable from another	frontend	Procedure	Start of wrong SW version	none	skill	Throughput degradation (affected web server unable to process client requests)	4
wrong_shutdown_apache	wrong_shutdown_frontend	[nagaraja04]	5	Shutdown both frontend web-servers at the same time	frontend	Procedure	Global misconfiguration	none	skill	Throughput degradation (both frontend servers inaccessible during restart)	4
incorrect_tomcat_noroot1	incorrect_tomcat_noroot	[nagaraja04]	5	Tomcat started incorrectly (forgot to obtain root privileges before starting) on app server	middle tier	Procedure	Incorrect restart	none	skill	Throughput degradation (Tomcat silently died on new app server)	4
mysql_nopass_root	mysql_nopass_root	[nagaraja04]	5	No password set up for MySQL root user	backend	Configuration	Local misconfiguration	none	rule	Security vulnerability	4
mysql_wrong_privileges	wrong_privileges_insufficient	[nagaraja04]	5	MySQL user not given necessary privileges	backend	Configuration	Global misconfiguration	none	rule	Service inaccessible (all Tomcat threads blocked)	4
wrong_apache2	wrong_apache	[nagaraja04]	5	Launched Apache from the wrong distribution while restarting the service	frontend	Procedure	Start of wrong SW version	none	skill	Service inaccessible (both frontend servers unable to process client requests)	4
wrong_db_disk	wrong_db_disk	[nagaraja04]	5	Installed the DB on a slow disk	backend	Procedure	Wrong choice of HW component	none	rule	Throughput degradation (limited capacity)	4
db_deployment_script	db_deployment_script	[oliveira06]	5	Bugs in the scripts for deploying DB changes	backend	Procedure	Deployment problems	none			4
deploy_schema_nochange	db_schema	[oliveira06]	5	Forgot to change the schema of the online DB before deploying a new or changed application	backend	Procedure	Deployment problems	none	knowledge	Fatal SQL errors	4
deploy_accidental_changes	deploy_accidental_changes	[oliveira06]	6	Accidentally propagated the changes made to the database in the testing environment	backend	Procedure	Deployment problems	none	rule		4
deploy_wrong_changes	deploy_wrong_changes	[oliveira06]	6	Inappropriate changes directly to the online database	backend	Procedure	Deployment problems	none	rule		4
deploy_index_nochange	deploy_index_nochange	[oliveira06]	6	Forgot to reapply indexes in the production database	backend	Procedure	Deployment problems	none	rule		4
wrong_schema_compile	db_schema	[oliveira06]	6	Applications compiled agains the wrong schema are deployed online	backend	Procedure	Deployment problems	none	knowledge	Fatal SQL errors	4
performance_wrong_schema	db_schema	[oliveira06]	6	Inappropriate database-structure design	backend	Procedure	Deployment problems	none	knowledge	Poor performance to applications or users	4

wrong_index	wrong_index	[oliveira06]	6	Inappropriate indexing scheme	backend	Procedure	Deployment problems	none	knowledge	Poor performance to applications or users	4
wrong_schema	db_schema	[oliveira06]	6	Incorrect database design (e.g. duplicated identity columns; columns too small to hold data)	backend	Procedure	Deployment problems	none	knowledge	Deadlocks; etc.	4
wrong_privileges_insufficient	wrong_privileges_insufficient	[oliveira06]	6	Insufficient acccess privileges granted to users and applications	backend	Configuration	Global misconfiguration	none	rule	Inability to access the whole or parts of the) database	4
wrong_privileges_excessive	wrong_privileges_excessive	[oliveira06]	6	Excessive access privileges granted to some users or applications	backend	Configuration	Global misconfiguration	none	rule	Security vulnerability	4
no_space	no_space	[oliveira06]	6	Disk space exhaustion	backend	Procedure	Space problems	none	rule		4
tablespace_full	tablespace_full	[oliveira06]	6	Tablespace problems	backend	Procedure	Space problems	none	rule		4
wrong_shutdown_dbms	wrong_shutdown_db	[oliveira06]	7	Incorrectly shut down the database	backend	Procedure	General maintenance problems	none	skill	Database inaccessible to the application servers	4
no_replication	no_replication	[oliveira06]	7	Forgot to restart the DBMS replication	backend	Procedure	General maintenance problems	none	skill		4
no_space_backup	no_space_backup	[oliveira06]	7	Insufficient backup space	backend	Procedure	General maintenance problems	none	rule		4
apache_42975	apache_42975	Apache Bugzilla	42975	ProxyPassReverse not handling relative redirects properly	frontend	Procedure	Other error	none	knowledge	Incorrect functionality	4
performance_tuning	performance_tuning	[oliveira06]	6	Erroneous performance tuning	backend	Configuration	Deployment problems	semantic	knowledge	Poor performance to applications or users	2
dbms_misconfiguration	dbms_misconfiguration	[oliveira06]	7	Misconfigured DBMS	backend	Configuration	General maintenance problems	semantic	knowledge	Could not restart DBMS	2
apache_41979	apache_41979	Apache Bugzilla	41979	Load Balancer Manager Web Client Is Blank	frontend	Configuration	Other error	semantic	knowledge	Incorrect functionality	2
apache_42605	apache_42605	Apache Bugzilla	42605	HTTPS-URL with special-port requesting a directory (without trailing slash) is rewritten to HTTPS-URL requesting a directory with trailing slash, BUT the specified port is missing	frontend	Configuration	Parameter tuning	semantic	knowledge	Incorrect functionality	2
apache_42627	apache_42627	Apache Bugzilla	42627	Unable to authenticate using authz-ldap require group	middle tier	Configuration	Other error	semantic	knowledge	Incorrect functionality	2
apache_42709	apache_42709	Apache Bugzilla	42709	.htaccess is viewable by browser after login validation	middle tier	Configuration	Other error	semantic	knowledge	Incorrect functionality	2
apache_42751	apache_largefile	Apache Bugzilla	42751	CIFS mounted filesystems do not transmit files	middle tier	Configuration	Parameter tuning	semantic	knowledge	Incorrect functionality	2
apache_43232	apache_largefile	Apache Bugzilla	43232	error while transmitting file over 64K	middle tier	Configuration	Parameter tuning	semantic	knowledge	Incorrect functionality	2
apache_41751	apache_41751	Apache Bugzilla	41751	Keepalive connections keep children tied up even if new requests starve		Configuration	Parameter tuning	semantic	knowledge	Performance degradation	2
apache_41358	apache_41358	Apache Bugzilla	41358	No DSO works: ./configure --enable-mods-shared doesn't work, neither does apxs, neither does ./configure --enable-proxy=shared.	frontend	Procedure	Build	none	rule	Error message	3
apache_42285	apache_42285	Apache Bugzilla	42285	mod_authnz_ldap reports [Can't contact LDAP server]	middle tier	Procedure	Environmental conflict	none	rule	Error message	3
apache_42332	apache_42332	Apache Bugzilla	42332	GDBM not supported in 2.2.4?	middle tier	Procedure	Build	none	rule	Error message	3
apache_42556	apache_42556	Apache Bugzilla	42556	Apache with LDAP support segfaults on Solaris 9 with LDAP	middle tier	Procedure	Environmental conflict	none	rule	Crash	3
apache_43328	apache_43328	Apache Bugzilla	43328	mod_authnz_ldap does not compile	middle tier	Configuration	Build	none	rule	Error message	3
apache_43518	incorrect_apache_port	Apache Bugzilla	43518	no listening sockets available		Procedure	Environmental conflict	none	rule	Error message	3
apache_43523	apache_43523	Apache Bugzilla	43523	Compatibility with last svn dav and authz .so modules...	middle tier	Procedure	Build	none	rule	Error message	3
apache_43945	apache_43945	Apache Bugzilla	43945	restart lockup		Procedure	Build	none	rule	Hang	3
apache_43986	apache_43986	Apache Bugzilla	43986	Multiple compile errors in mod_disk_cache.c		Procedure	Build	none	rule	Error message	3