Understanding and Dealing with Operator Mistakes in Internet Services *

Kiran Nagaraja, Fábio Oliveira, Ricardo Bianchini, Richard P. Martin, Thu D. Nguyen
Department of Computer Science
Rutgers University, Piscataway, NJ 08854
{*knagaraj, fabiool, ricardob, rmartin, tdnguyen*}*@cs.rutgers.edu*

Abstract

Operator mistakes are a significant source of unavailability in modern Internet services. In this paper, we first characterize these mistakes by performing an extensive set of experiments using human operators and a realistic three-tier auction service. The mistakes we observed range from software misconfiguration, to fault misdiagnosis, to incorrect software restarts. We next propose to validate operator actions before they are made visible to the rest of the system. We demonstrate how to accomplish this task via the creation of a validation environment that is an extension of the online system, where components can be validated using real workloads before they are migrated into the running service. We show that our prototype validation system can detect 66% of the operator mistakes that we have observed.

1 Introduction

Online services, such as search engines, e-mail, workgroup calendars, and music juke-boxes are rapidly becoming the supporting infrastructure for numerous users' work and leisure. Increasingly, these services are comprised of complex conglomerates of distributed hardware and software components, including front-end devices, numerous kinds of application packages, authenticators, loggers, databases, and storage servers.

Ensuring high availability for these services is a challenging task. First, frequent hardware and software upgrades keep these systems constantly evolving. Second, this evolution and the complexity of the services imply a large number of unforeseen interactions. Third, component failure is a common occurrence, since these services are typically based on commodity components for fast deployment and low cost. Given these factors, it is not surprising that service failures occur frequently [12, 19, 20].

In this paper, we characterize and alleviate one significant source of service failures, namely *operator mistakes*, in the context of cluster-based Internet services. Several studies have shown that the percentage of service failures attributable to operator mistakes has been increasing over the last decade [12, 17, 20]. A recent study of three commercial services showed that operator mistakes were responsible for 19-36% of the failures, and, for two of the services, were the dominant source of failures and the largest contributor to time to repair [19]. An older study of Tandem systems also found that operator mistakes were a dominant reason for outages [11].

Our work begins with a set of live operator experiments that explore the nature of operator mistakes and their impact on the availability of a three-tier auction service [21]. In each experiment, an operator must either perform a scheduled maintenance task or a diagnose-and-repair task. The first category encompasses tasks such as upgrading software, upgrading hardware, and adding or removing system components. The second category encompasses experiments during which we inject a fault into the service and ask the operator to discover and fix the problem.

The operator experiments do *not* seek to cover all possible operator tasks or to achieve a complete statistical characterization of operator behavior. Rather, our goal is to characterize some of the mistakes that can occur during common operator tasks, and to gather detailed traces of operator actions that can be used to evaluate the effectiveness of techniques designed to either prevent or to mitigate the impact of operator mistakes. We are continuing our experiments to cover a wider range of operator tasks and to collect a larger sampling of behaviors.

So far, we have performed 43 experiments with 21 volunteer operators with a wide variety of skill levels. Our results show a total of 42 mistakes, ranging from software configuration, to fault misdiagnosis, to soft-

*This research was partially supported by NSF grants #EIA-0103722, #EIA-9986046, and #CCR-0100798.

ware restart mistakes. Configuration mistakes of different types were the most common with 24 occurrences, but incorrect software restarts were also common with 14 occurrences. A large number of mistakes (19) led to a degradation in service throughput.

Given the large number of mistakes the operators made, we next propose that services should *validate* operator actions before exposing their effects to clients. The key idea is to check the correctness of operator actions in a *validation environment* that is an extension of the online system. In particular, the components under validation, called *masked* components, should be subjected to realistic (or even live) workloads. Critically, their configurations should not have to be modified when transitioning from validation to live operation.

To demonstrate our approach and evaluate its efficacy, we have implemented a prototype validation framework and modified two applications, a cooperative Web server and our three-tier auction service, to work within the framework. Our prototype currently includes two validation techniques, trace-based and replica-based validation. Trace-based validation involves periodically collecting traces of live requests and replaying the trace for validation. Replica-based validation involves designating each masked component as a "mirror" of a live component. All requests sent to the live components are then duplicated and also sent to the mirrored, masked component. Results from the masked components are compared against those produced by the live component.

We evaluate the effectiveness of our validation approach by running a wide range of experiments with our prototype: (1) microbenchmarks that isolate its performance overhead; (2) experiments with human operators; (3) experiments with mistake traces; and (4) mistake-injection experiments. From the microbenchmarks, we find that the overhead of validation is acceptable in most cases. From the other experiments, we find that our prototype is easy to use in practice, and that the combination of trace and replica-based validation is effective in catching a majority of the mistakes we have seen. In particular, using detailed traces of operator mistakes, we show that our prototype would have detected 28 out of the 42 mistakes observed in our operator experiments.

In summary, we make two main contributions:

- We present detailed data on operator behavior during a large set of live experiments with a realistic service. Traces of all our experiments are available from `http://vivo.cs.rutgers.edu/`. This contribution is especially important given that actual data on operator mistakes in Internet services is not publicly available, due to commercial and privacy considerations. We also analyze and categorize the reasons behind the mistakes in detail.
- We design and implement a prototype validation framework that includes a realistic validation environment for dealing with operator mistakes. We demonstrate the benefits of the prototype through an extensive set of experiments, including experiments with actual operators.

We conclude that operators make mistakes even in fairly simple tasks (and with plenty of detailed information about the service and the task itself). We conjecture that these mistakes are mostly a result of the complex nature of modern Internet services. In fact, a large fraction of the mistakes cause problems in the interaction between the service components in the actual processing of client requests, suggesting that the realism derived from hosting the validation environment in the online system itself is critical. Given our experience with the prototype, we also conclude that validation should be useful for real commercial services, as it is indeed capable of detecting several types of operator mistakes.

The remainder of the paper is organized as follows. The next section describes the related work. Section 3 describes our operator experiments and their results. Section 4 describes the details of our validation approach and prototype implementation, and compares our approach with offline testing and undo in the context of the mistakes we observed. Section 5 presents the results of our validation experiments. Finally, Section 6 concludes the paper.

2 Related Work

Only a few papers have addressed operator mistakes in Internet services. The work of Oppenheimer *et al.* [19] considered the universe of failures observed by three commercial services. With respect to operators, they broadly categorized their mistakes, described a few example mistakes, and suggested some avenues for dealing with them. Here, we extend their work by describing all of the mistakes we observed in detail and by designing and implementing a prototype infrastructure that can detect a majority of the mistakes.

Brown and Patterson [6] have proposed "undo" as a way to rollback state changes when recovering from operator mistakes. Brown [5] has also performed experiments in which he exposed human operators to an implementation of undo for an email service hosted by a single node. We extend his results by considering a more complex service hosted by a cluster. Furthermore, our validation approach is orthogonal to undo in that we hide operator actions from the live service until they have been validated in a realistic validation environment. We discuss undo further in Section 4.6.

A more closely related technique is "offline testing" [3]. Our validation approach takes offline testing a step further by operating on components in a validation environment that is an extension of the live service. This allows us to catch a larger number of mistakes, as we discuss in Section 4.6.

The Microvisor work by Lowell *et al.* [16] isolates online and maintenance parts of a single node, while keeping the two environments identical. However, their work cannot be used to validate the interaction among components hosted on multiple nodes.

Our trace-based validation is similar in flavor to various fault diagnosis approaches [2, 10] that maintain statistical models of "normal" component behavior and dynamically inspect the service execution for deviations from this behavior. These approaches typically focus on the data flow behavior across the systems components, whereas our trace-based validation inspects the actual responses coming from components and can do so at various semantic levels. We also use replica-based validation, which compares the responses directly to those of "correct" live components.

Replication-based validation has been used before to tolerate Byzantine failures and malicious attacks, e.g. [8, 9, 13]. In this context, replicas are a permanent part of the distributed system and validation is constantly performed through voting. In contrast, our approach focuses solely on dealing with operator mistakes, does not require replicas and validation during normal service execution, and thus can be simpler and less intrusive.

Other orthogonal approaches to dealing with operator mistakes or reducing operator intervention have been studied [1, 4, 14]. For example, Ajmani *et al.* [1] eliminate operator intervention in software upgrades, whereas Kiciman *et al.* [14] automate the construction of correctness constraints for checking software configurations.

3 Operator Actions and Mistakes

In this section, we analyze the maintenance and diagnose-and-repair experiments that we performed with human operators and our three-tier auction service. We start by describing the experimental setup, the actual experiments, and the operators who volunteered to participate in our study. After that, we detail each experiment, highlighting the mistakes made by the operators.

3.1 Experimental setup

Our experimental testbed consists of an online auction service modeled after EBay. The service is organized into three tiers of servers: Web, application, and database tiers. We use two machines in the first tier running the Apache Web server (version 1.3.27), five machines running the Tomcat servlet server (version 4.1.18) in the second tier and, in the third tier, one machine running the MySQL relational database (version 4.12). The Web server and application server machines are equipped with a 1.2 GHz Intel Celeron processor and 512 MB of RAM, whereas the database machine relies on a 1.9 GHz Pentium IV with 1 GB of RAM. All machines run Linux with kernel 2.4.18-14.

The service requests are received by the Web servers and may flow towards the second and third tiers. The replies flow through the same path in the reverse direction. Each Web server keeps track of the requests it sends to the application servers. Each application server maintains the soft state associated with the client sessions that it is currently serving. This state consists of the auctions of interest to the clients. All dynamic requests belonging to a session need to be processed by the same application server, thereby restricting load balancing. A heartbeat-based membership protocol is used to reconfigure the service when nodes become unavailable or are added to the cluster.

A client emulator is used to exercise the service. The workload consists of a number of concurrent clients that repeatedly open sessions with the service. Each client issues a request, receives and parses the reply, "thinks" for a while, and follows a link contained in the reply. A user-defined Markov model determines which link to follow. During our experiments, the overall load imposed on the system is 200 requests/second, which is approximately 35% of the service's maximum achievable throughput. The code for the service and client emulator is publicly available from the DynaServer project [21] at Rice University.

Another important component of our experimental setup is a monitoring infrastructure that includes a shell that records and timestamps every single command (and the corresponding result) executed by the operator. The infrastructure also measures the system throughput on-the-fly, presenting it to the operator so that he/she can visually assess the impact of his/her actions on the system performance.

3.2 Experiments with operators

Our experiments can be categorized as either scheduled maintenance tasks or diagnose-and-repair tasks. Table 1 summarizes the classes of experiments.

Before having the operator interact with the system, we provide him/her with conceptual information on the system architecture, design, and interface. We convey this information verbally and through a graphical representation of the system which can be consulted at any time during an experiment. We also give the operator two

Task Category	Subcategory
Scheduled maintenance	Node addition
	Data migration
	Software upgrade
Diagnose-and-repair	Software misconfiguration
	Application crash/hang
	Hardware fault

Table 1: *Categories of experiments.*

sets of written instructions: general directions concerning the system interface and specific instructions about the task the operator will be doing. The operator is allowed to refer to both sets during the experiment.

A total of 21 people with varying skill levels volunteered to act as operators in our experiments: 14 graduate students, 2 operations staff members, and 5 professional programmers. The students and staff are from our own department; one of the staff members is the system administrator for one of our large clusters, and the other is the database administrator of our department. Two of the programmers used to work as system administrators, and four of them currently work for the Ask Jeeves commercial search engine.

To investigate the distribution of operator mistakes across different skill levels, we divide our operators into three categories: novice, intermediate, and expert. We deem the staff members and three of the professional programmers to be experts based on their experience in system administration. The remaining two programmers were classified as intermediate. Finally, we asked the graduate students to complete a questionnaire designed to assess their operation experience. Based on their responses, we ended up with five experts, five intermediates, and eleven novices. In Section 3.9, we discuss the breakdown of mistakes across these operator categories.

We gave the novice operators a "warm up" task involving the addition of a new Web server to the system to give them the opportunity to understand the system configuration and tier-to-tier communication issues, as well as crystallize the information we had conveyed orally. For this task, we provided very detailed instructions. (In our study, we do not take the mistakes made in this warm up task into consideration.)

All sets of instructions, the questionnaire, and the operator behavior data we collected are available at `http://vivo.cs.rutgers.edu/`.

3.3 Maintenance task 1: add an application server

In this experiment we ask the operator to add a Tomcat server to the second tier. In a nutshell, the operator is supposed to copy the Tomcat binary distribution from any machine in the second tier to the specified new machine and configure it properly, so that it can exchange information with the database in the third tier. In addition, it is necessary to correctly reconfigure and restart the Web servers for the newly added Tomcat server to actually receive and process requests. The experiment is set up in such a way that the system has enough resources to handle the load imposed by the client emulator; hence, the new Tomcat server does not imply any increase in throughput.

This experiment has been conducted with eight novice operators, four intermediates, and two experts, with an average time per run of one hour. Two of the operators were completely successful in that they did not make any configuration mistakes or affect the system's ability to service client requests more than what was strictly necessary. On the other hand, the other operators made mistakes with varying degrees of severity. The next few paragraphs discuss these mistakes.

Apache misconfigured. This was the most common mistake. We recognized four different flavors of it, all of them affecting the system differently. In the least severe misconfiguration, three novice operators did not make all the needed modifications to the Apache configuration file. In particular, they added information about the new machine to the file, but forgot to add the machine's name to the very last line of file, which specifies the Tomcat server names. As a result, even though Tomcat was correctly started on the new machine, client requests were never forwarded to it. The operators who made this mistake either did not spend any time looking at the Apache log files to make sure that the new Tomcat server was processing requests or they analyzed the wrong log files. Although this misconfiguration did not affect the system performance immediately, it introduced a latent error.

Another flavor of Apache misconfiguration was more subtle and severe in terms of performance impact. One novice operator introduced a syntax error when editing the Apache configuration file that caused the module responsible for forwarding requests to the second tier (`mod_jk`) to crash. The outcome was the system's inability to forward requests to the second tier. The operator noticed the problem by looking desperately at our performance monitoring tool, but could not find the cause after 10 minutes trying to do so. At that point, he gave up and we told him what the problem was.

One more mistake occurred when one expert and one novice modified the configuration file but left two identical application server names. This also made `mod_jk` crash and led to a severe throughput drop: a decrease of about 50% when the mistake affected one of the Web servers, and about 90% when both Web servers were compromised. The operators who made this mistake were not able to correct the problem for 20 minutes on

average. After this period, they decided not to continue and we showed them their mistakes.

Finally, one intermediate operator forgot to modify the Apache configuration file altogether to reflect the addition of the new application server. This mistake resulted in the inability of Apache to forward requests to the new application server. The operator was able to detect his mistake and fix the problem in 24 minutes.

Apache incorrectly restarted. In this case, one intermediate operator reconfigured one Apache distribution and launched the executable file from another (there were two Apache distributions installed on the first-tier machines). This mistake made the affected Web server become unable to process any client requests.

Bringing down both Web servers. In a few experiments, while reconfiguring Apache to take into account the new application server, two novice and three intermediate operators unnecessarily shutdown both Web servers at the same time and, as a consequence, made the whole service unavailable.

Tomcat incorrectly started. One novice and one expert were unable to start Tomcat correctly. In particular, they forgot to obtain root privileges before starting Tomcat. The expert operator started Tomcat multiple times without killing processes remaining from the previous launches. This scenario led to Tomcat silently dying. To make matters worse, since the heartbeat service — which is a separate process — was still running, the Web servers continued forwarding requests to a machine unable to process them. The result was substantially degraded throughput. The operators corrected this mistake in 22 minutes on average.

3.4 Maintenance task 2: upgrade the database machine

The purpose of this experiment is to migrate the MySQL database from a slow machine to a powerful one, which is equipped with more memory, a faster disk, and a faster CPU. (Note that we use the fast machine in all other experiments.) Because the database machine is the bottleneck of our testbed and the system is saturated when the slow machine is used, the expected outcome of this experiment is higher service throughput.

This experiment involves several steps: (1) compile and install MySQL on the new machine; (2) bring the whole service down; (3) dump the database tables from the old MySQL installation and copy them to the new machine; (4) configure MySQL properly by modifying the `my.cnf` file; (5) initialize MySQL and create an empty database; (6) import the dumped files into the empty database; (7) modify the relevant configuration files in all application servers so that Tomcat can forward requests to the new database machine; and (8) start up MySQL, all application servers, and Web servers.

Four novices, two intermediates, and two experts performed this task; the average time per run was 2 hours and 20 minutes. We next detail the mistakes observed.

No password set up for MySQL root user. One novice operator failed to assign a password to the MySQL root user, during MySQL configuration. This mistake led to a severe security vulnerability, allowing virtually anyone to execute any operation on the database.

MySQL user not given necessary privileges. As part of the database migration, the operators need to ensure that the application servers are able to connect to the database and issue the appropriate requests. This involves reconfiguring Tomcat to forward requests to the new database machine and granting the proper privileges to the MySQL user that Tomcat uses to connect to the database. One novice and one expert did not grant the necessary privileges, preventing all application servers from establishing connections to the database. As a result, all Tomcat threads eventually got blocked and the whole system became unavailable. The expert managed to detect and correct the problem in 45 minutes. The novice did not even try to identify the problem.

Apache incorrectly restarted. One intermediate operator launched Apache from the wrong distribution, while restarting the service. Again, this mistake caused the service to become completely unavailable. It took the operator 10 minutes to detect and fix the mistake.

Database installed on the wrong disk. The powerful machine had two disks: a 15K RPM SCSI disk and a 7200 RPM IDE disk. Given that the database machine was known to be the bottleneck of our system and database migration was needed so that the service could keep up with the load imposed by the emulated clients, the operators should not have hesitated to install MySQL on the faster SCSI disk. One novice operator installed the database on the slow disk, limiting the throughput that can be achieved by the service. The operator never realized his mistake.

3.5 Maintenance task 3: upgrade one Web server

In this experiment, the operators are required to upgrade Apache from version 1.3.27 to version 2.0.49 on one machine. In a nutshell, this involves downloading the Apache source code from the Web, compiling it, configuring it properly, and integrating it into the service.

Two intermediate and three expert operators participated in this maintenance experiment. The average time per run was about 2 hours. We describe the observed mistakes next.

Apache misconfigured. Before spawning the Web server processes, Apache version 2.0.49 automatically invokes a syntax checker that analyzes the main configuration file to make sure that the server will not be started in the event of configuration syntax errors. This feature is extremely useful to avoid exposing operator mistakes to the live system. In our experiments, the syntax checker actually caught three configuration mistakes involving the use of directives no longer valid in the newer Apache version. However, as the checker is solely concerned with syntax, it did not catch some other configuration mistakes. One expert launched Apache without specifying in the main configuration file how Apache should map a given URL to a request for a Tomcat servlet. This misconfiguration led to the inability of the upgraded Apache to forward requests to the second tier, causing degraded throughput. The operator fixed this problem after 10 minutes of investigation.

In addition, a latent error resulted from two other misconfigurations. Two experts and one intermediate configured the new Apache server to get the HTML files from the old Apache's directory tree. A similar mistake was made with respect to the location of the heartbeat service program. The latent error would be activated if someone removed the files belonging to the old distribution.

Yet another mistake occurred when one expert correctly realized that the heartbeat program should be executed from the new Apache's directory tree, but incorrectly specified the path for the program. In our setup, the heartbeat program is launched by Apache. Because of the wrong path, `mod_jk` crashed when the new Apache was started. This made the new server unable to process requests for dynamic content, resulting in throughput degradation. The operator was able to fix this problem in 13 minutes.

3.6 Diagnose-and-repair task 1: Web server misconfiguration and crash

To observe the operator behavior resulting from latent errors that become activated, we performed experiments in which an Apache server misconfiguration and later crash are injected into the system. This sequence of events mimics an accidental misconfiguration or corruption of the configuration file that is followed by the need to restart the server.

In more detail, the system starts operating normally. At some point after the experiment has begun, we modify the configuration file pertaining to `mod_jk` in one of the Apache servers, so that a restart of the server will cause a segmentation fault. Later, we crash the same server to force the operator to restart it. As soon as the server is abnormally terminated, the throughput decreases to half of its prior value. The operators' task is to diagnose and fix the problem, so that normal throughput can be regained.

This experiment was presented to three novices, three intermediates, and two experts, and the average time per run was 1 hour and 20 minutes. Two operators were able to both understand the system's malfunctioning and fix it in about 1 hour and 15 minutes. All operators — even the successful ones — made some mistakes, most of which aggravated the problem. All of the mistakes were caused by misdiagnosing the source of the service malfunction.

Misdiagnosis. Due to misdiagnosis, operators of all categories unnecessarily modified configuration files all over the system, which, in one case, caused the throughput to drop to zero. The apparent reason for such behavior was the fact that some operators were tempted to literally interpret the error messages appearing in the log files, instead of reasoning about what the real problem was. In other words, when reading something like "Apache seems to be busy; you should increase the *MaxClients* parameter..." in a log file, some operators performed the suggested action without further reasoning.

We also noticed the mistake of starting the wrong Apache distribution (as previously discussed) made by one novice and one intermediate, severely degrading the throughput or making it drop to zero. A couple of operators even suggested the replacement of hardware components, as a result of incorrectly diagnosing the problem as a disk or a memory fault.

3.7 Diagnose-and-repair task 2: application server hang

In this experiment, we inject another kind of fault: we force Tomcat to hang on three second-tier machines. The system is working perfectly until we perturb it.

We conducted this experiment with two novices, one intermediate, and one expert operator. All operators were able to detect the fault and fix the problem after 1 hour and 30 minutes on average. However, we noticed some mistakes as discussed next.

Tomcat incorrectly restarted. One novice operator restarted one of the two working servlet servers without root privileges, causing it to crash. This caused the service to lose yet another servlet server and the remaining one became overloaded. The operator was only able to detect the crashed Tomcat server 20 minutes later.

Database unnecessarily restarted. While trying to diagnose the problem, one novice operator unnecessarily restarted the database server. As the database machine is not replicated, bringing it down results in the system's inability to process most requests.

MySQL denied write privileges. One intermediate operator, while trying to diagnose the problem, decided to thoroughly verify the MySQL files. The operator in-

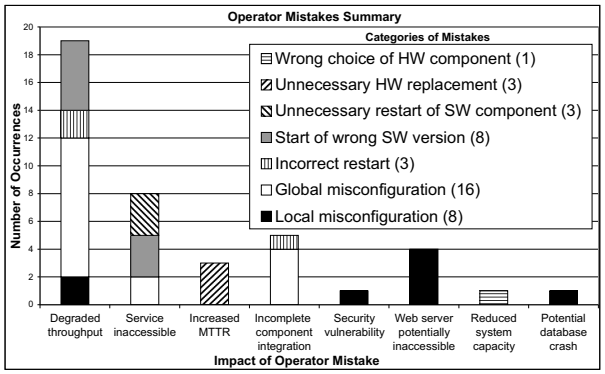

Figure 1: *Operator mistakes and their impact.*

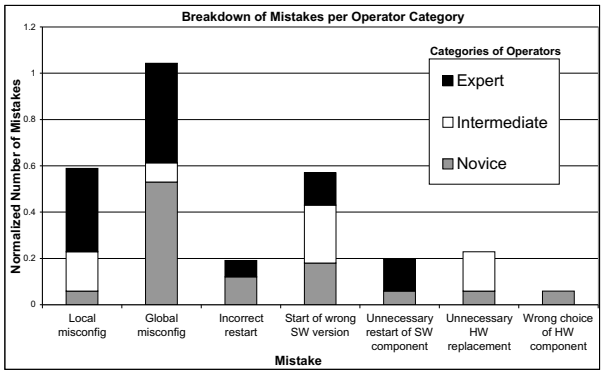

Figure 2: *Operator mistakes per operator category.*

advertently write-protected the whole MySQL directory tree, leading to MySQL's inability to write to the tables. This mistake did not cause any immediate effect because the files containing the tables had already been opened by the database server. However, this mistake led to a latent error that would arise if, for some reason, the database had to be restarted.

3.8 Diagnose-and-repair task 3: disk fault in the database machine

In this experiment, we use the Mendosus fault-injection and network-emulation tool [15] to force a disk timeout to occur periodically in the database machine. The timeouts are injected according to an exponential inter-arrival distribution with an average rate of 0.03 occurrences per second. Since the database is the bottleneck, the disk timeouts substantially decrease the service throughput.

Four experts participated in this experiment. Of these four operators, three were unable to discover the problem after interacting with the system for 2 hours on average, and the other one correctly diagnosed the fault in 34 minutes. Throughout their interaction with the service, the unsuccessful operators made mistakes caused by misdiagnosing the real root of the problem.

Misdiagnosis. Two operators ended up diagnosing the fault as an "intermittent network problem" between the second and third tiers. Before the operators reached that conclusion, we had observed other incorrect diagnoses on their part such as DoS attack, Tomcat misconfiguration, and lack of communication between the first and second tiers. The other operator was suspicious of a MySQL misconfiguration and tried to adjust some parameters of the database and subsequently restarted it.

Under the influence of error messages reported in the log files, one operator changed, in two application server machines, the port which Tomcat was using to receive requests from Apache; as a result, the affected application servers became unreachable. The other two operators looked at the main kernel log on the database machine and saw several messages logged by the SCSI driver reporting the disk malfunction. Unfortunately, they ignored such messages and did not even suspect that the disk was misbehaving.

3.9 Summary

Figures 1 and 2 summarize our findings. The X-axis in Figure 1 indicates the effects of the different operator mistakes, whereas the stacked bars show the number of occurrences of the mistake categories listed in the legend. The legend also shows the number of mistakes in each mistake category. In the figure, "incomplete component integration" refers to scenarios in which an added component is not seen by other components, "wrong choice of HW component" refers to installing the database on a slow disk, "unnecessary HW replacements" refers to misdiagnosing service malfunction as a hardware problem, and "unnecessary restart of SW component" refers to restarts of the database server. Overall, we observed 42 mistakes. In some cases, a single operator made more than one mistake during a particular experiment.

As we indicated before, misconfiguration was the most frequent mistake in our experiments. In Figure 1, we distinguish between local and global misconfiguration mistakes. Global misconfiguration refers to inconsistencies in one or more configuration files compromising the communication between system components, whereas local misconfiguration refers to misconfigurations that affect only one component of the system.

A local misconfiguration is a configuration mistake that caused Tomcat to crash, led to a security vulnerability, or could potentially prevent Apache from servicing requests. Global misconfigurations involve mistakes that: (1) prevented one or both Web servers from either forwarding requests to any application server machine, or sending requests to a newly added application server; (2) prevented the application servers from establishing

connections with the database server; (3) prevented one or both Web servers from processing requests; and (4) led one or both Web servers to forward requests to a non-existing application server.

In Figure 2, we show the distribution of mistakes across our three operator categories. Given that no operator took part in all types of experiments, we normalized the number of mistakes by dividing it by the total number of experiments in which the corresponding operator category participated. As the figure illustrates, experts made mistakes (especially misconfigurations) in a significant fraction of their experiments. The reason for this counter-intuitive result is that the hardest experiments were performed mostly by the experts, and those experiments were susceptible to local and global misconfiguration.

As we discuss in detail later, our validation approach would have been able to catch the majority (66% or 28 mistakes) of the 42 mistakes we observed. The remaining 14 mistakes, including unnecessary software restarts and unnecessary hardware replacements, were made by expert (6 mistakes), intermediate (4 mistakes), and novice (4 mistakes) operators.

4 Validation

Given that even expert operators make many mistakes, we propose that operator actions should be *validated* before their effects are exposed to end users. Specifically, we build a validation infrastructure that allows components to be validated in a slice of the online system itself, rather than being tested in a separate offline environment.

We start this section with an overview of our proposed validation approach. Then, we describe a prototype implementation and our experience in modifying the three-tier auction service to include validation. We have also implemented a validation framework for the PRESS clustered Web server [7]. PRESS is an interesting counter-point to our multithreaded auction service as it is a single-tier, event-based server. Our earlier technical report [18] discusses this implementation. Finally, we close the section with a discussion of how operators can still make mistakes even with validation and with a comparison of validation with offline testing and undo.

4.1 Overview

A validation environment should be closely tied to the online system for three reasons: (1) to avoid latent errors that escape detection during validation but become activated in the online system, because of differences between the validation and online environments; (2) to load components under validation with as realistic a workload as possible; and (3) to enable operators to bring validated components online without having to change any of the components' configurations, thereby minimizing the chance of new operator mistakes. On the other hand, the components under validation, which we shall call *masked* components for simplicity, must be *isolated* from the live service so that incorrect behaviors cannot cause service failures.

To meet the above goals, we divide the cluster hosting a service into two logical slices: an online slice that hosts the live service and a validation slice where components can be validated before being integrated into the live service. Figure 3 shows this validation architecture in the context of the three-tier auction service. To protect the integrity of the live service without completely separating the two slices (which would reduce the validation slice to an offline testing system), we erect an isolation barrier between the slices but introduce a set of connecting *shunts*. The shunts are one-way portals that duplicate requests and replies (i.e., inputs and outputs) passing through the interfaces of the components in the live service. Shunts either log these requests and replies or forward them to the validation slice. Shunts can be easily implemented for either open or proprietary software, as long as the components' interfaces are well-defined.

We then build a validation harness consisting of *proxy components* that can be used to form a virtual service around the masked components as shown by the dashed box in Figure 3. Together, the virtual service and the duplication of requests and replies via the shunts allow operators to validate masked components under realistic workloads. In particular, the virtual service either replays previously recorded logs or accepts forwarded duplicates of live requests and responses from the shunts, feeds appropriate requests to the masked components, and verifies that the outputs of the masked components meet certain validation criteria. Proxies can be implemented by modifying open source components or wrapping code around proprietary software with well-defined interfaces.

Finally, the validation harness uses a set of *comparator functions* to test the correctness of the masked components. These functions compute whether some set of observations of the validation service match a set of criteria. For example, in Figure 3, a comparator function might determine if the streams of requests and replies going across the pair of connections labeled A and those labeled B are similar enough (A to A and B to B) to declare the masked Web server as working correctly. If any comparison fails, an error is signaled and the validation fails. If after a threshold period of time all comparisons match, the component is considered validated.

Given the above infrastructure, our approach is conceptually simple. First, the operator places the components to be worked on in the validation environment, effectively masking them from the live service. The oper-

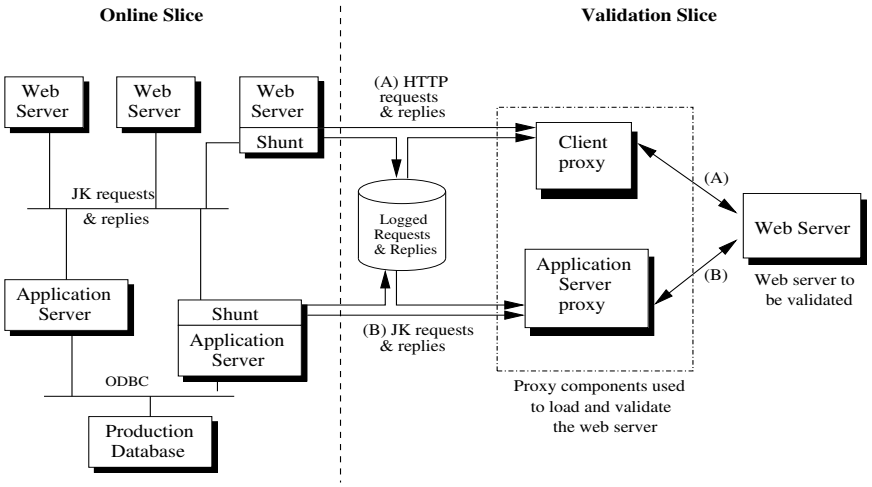

Figure 3: *The three-tier auction service with validation. In this particular case, a single component, a Web server, is being validated inside the validation slice. The validation harness uses one or more client proxies to load the Web server and one or more application server proxies to field requests for dynamic content from the Web server.*

ator then acts on the masked components just as he/she would in the live service. Next, the operator instructs the validation harness to surround the masked components with a virtual service, load the components, and check their correctness. If the masked components pass this validation, the operator calls a migration function that fully integrates the components into the live service.

4.2 Slice isolation

A critical challenge in building a validation infrastructure is how to isolate the slices from each other yet allow validated components to be migrated to the live service without requiring any changes to their internal state and configuration parameters. Our current implementation achieves this isolation and transparent migration at the granularity of an entire node by running nodes over a virtual network created using Mendosus [15].

Given the virtual network, it is fairly easy to impose the needed isolation: Mendosus was designed to inject network faults that would partition a network. Thus, we simply instruct Mendosus to partition our system into two parts to isolate the two slices from each other. Mendosus runs on each node, and, when enforcing a network partition, drops all packets, including multicast packets, that would normally flow between nodes separated by the partition. This enforced partition means that nodes in the validation slice can see each other but not the ones in the online slice and vice-versa. (To tunnel through this barrier, the shunts forward information to special nodes that have the privilege to bypass the network partition.)

Our virtual network then allows a node to be migrated between the slices without requiring any changes to the node's network configurations, as long as the software components comprising the service can dynamically discover each other and automatically adjust the service configuration to include all components running in a slice, which is a characteristic of all production-level services. This transparent network-level migration is particularly critical for detecting the global misconfiguration mistakes described in Section 3.9. Migrating live components without modifying their internal state is more difficult. We detail how we accomplish this migration below.

4.3 Validation strategies

An inherently difficult problem for validation is how to drive masked components with realistic workloads and validate their correctness. Consider the validation of a single component. One possible approach is to create a demanding workload to stress-test the component. Such an approach lends itself to *trace-based* techniques, where the requests and replies passing through the shunts of an equivalent live component are logged and later replayed. During the replay, the logged replies can be compared to the replies produced by the masked component. A second approach, *replica-based*, is to use the current offered load on the live service, where requests passing through the shunts of an equivalent live component are duplicated and forwarded in real-time to the validation harness to drive the masked component. The shunts also capture the replies generated by the live component and forward them to the harness, which compares them against the replies coming from the masked component.

The core differences between the two approaches are the assumptions about the request stream and the connections between components. For example, logged (or

even synthetic) request streams that exercise known difficult cases may be preferable to a light live load. On the other hand, replica-based validation may be necessary if the service or data sets have changed sufficiently that previously collected traces are no longer applicable.

In reality, these two strategies are not mutually exclusive; a particular validation might apply one or both approaches before concluding that a masked component is working properly. Further, we can use the same supporting infrastructure to implement both approaches: data collected by the shunts can either be saved to disk or forwarded directly to the validation slice. Thus, building both of these approaches in the same system is quite desirable, as long as there is sufficient storage bandwidth (trace collection) and network bandwidth (live forwarding). We explore the resource requirements of these approaches further in Section 5.1.

State management. An important issue related to the validation strategy is component state management. In our work, component state is any application-defined, externalizable data (either hard or soft), accrued during service execution that can affect subsequent responses. Specifically, our validation system faces two state management issues: (1) how to initialize a masked component with the appropriate internal state so that it can handle the validation workload correctly, and (2) how to migrate a validated component to the online slice without migrating state that may have accumulated during validation but is not valid for the live service.

The way we initialize the masked component depends on the validation strategy. In trace-based validation, a masked component is initialized with state recorded in the trace. We record this state before trace capture, but after we halt the component to be traced by temporarily buffering all incoming requests in the shunts and allowing all current requests to complete. In replica-based validation, the masked component is initialized with a copy of the state of the associated live component. We make the copy exactly as in trace-based validation, except that the state is forwarded to the masked component instead of being saved to disk.

In general, this strategy should be applicable to any component that supports the checkpointing of its state, including proprietary systems. However, care must be taken to avoid capturing state under overload conditions, when sufficient buffering may not exist to enable the momentary halting of the component. Further, it may be impossible to capture a large amount of state, such as that of a database, online. In this case, the component may need to be removed from active service before its state can be captured.

After validation, the operator can move a component holding only soft state to the online slice by restarting it and instructing Mendosus to migrate it to the online slice, where it will join the service as a fresh instance of that component type. For components with hard state, migration to the online slice may take two forms: (1) if the application itself knows how to integrate a new component of that type into the service (which may involve data redistribution), the component can be migrated with no state similar to components with only soft state; (2) otherwise, the operator must restart the component with the appropriate state before migrating it.

Multi-component validation. While the above discussion assumes the validation of a single component for simplicity, in general we need to validate the interaction between multiple components to address many of the global configuration mistakes described in Section 3. For example, when adding a new application server to the auction service, the Web servers' list of application servers must expand to include the new server. If this list is not updated properly, requests will not be forwarded to the new server after a migration, introducing a latent error. To ensure that this connection has been configured correctly, we must validate the existing Web servers and the new application server concurrently.

We call the above multi-component validation and currently handle it as follows. Suppose the operator needs to introduce a new component that requires changes to the configurations of existing live components, such as the adding of an application server. The operator would first introduce the new component to the validation slice and validate its correctness as discussed above. Next, each component from the live service whose configuration must be modified (so that it can interact properly with this new component) is brought into the validation slice one-by-one and the component pair is validated together to ensure their correct interoperability. After this validation, the existing component is migrated back into the online slice; the new component is only migrated to the online slice after checking its interactions with each existing component whose configurations had to be changed.

Note that there are thus at most two components under validation at any point in time in the above multi-component validation approach. In general, multi-component validation could be extended to include up to k additional components, but so far we have not found it necessary to have $k > 1$.

4.4 Implementing validation

Setting up the validation infrastructure involves modifying the online slice to be able to shunt requests and responses, setting up the harness composed of proxies within the validation slice, defining appropriate comparators for checking correctness, and implementing mechanisms for correct migration of components across

the two slices. In this section, we first discuss these issues and then comment on their implementation effort with respect to the auction service.

Shunts. We have implemented shunts for each of the three component types in the three-tier auction service. This implementation was straightforward because all inter-component interactions were performed using well-defined middleware libraries. Figure 3 shows that client requests and the corresponding responses are intercepted within the JK module on the Apache side. The current implementation does not intercept requests to static content, which represent a small percentage of all requests. The requests and responses to and from the database, via the Open DataBase Connectivity (ODBC) protocol, are intercepted in Tomcat's MySQL driver as SQL queries and their corresponding responses. In addition to duplicating requests and replies, we also tag each request/reply pair with a unique identifier. This ID is used by the validation harness to identify matching requests and responses generated by a masked component with the appropriate logged or forwarded requests and responses from the live system to which the comparator functions can be applied.

Validation harness. The validation harness needs to implement a service around the masked components in order to exercise them and check their functionalities. For example, to validate an application server in the auction service, the validation harness would need to provide at least one Web server and one database server to which the masked application server can connect.

One approach to building a service surrounding a masked component is to use a complete set of real (as opposed to proxy) components. This is reminiscent of offline testing, where a complete model of the live system is built. Although this approach is straightforward, it has several disadvantages. First, we would need to devote sufficient resources to host the entire service in the validation slice. Second, we would need a checkpoint of all service state, such as the content of the database and session state, at the beginning of the validation process. Finally, we would still need to fit the appropriate comparators into the real components.

To address the above limitations, we built lighter weight component proxies that interact with the masked component without requiring the full service. The proxies send requests to the masked component and check the replies coming from it. For services in which communicating components are connected using a common connection mechanism, such as event queues [7, 22], it is straightforward to realize the entire virtual service as collection of such queues in a single proxy. For heterogeneous systems like the auction service however, the tiers connect to each other using a variety of communication mechanisms. Thus, we have built different proxies representing the appropriate connection protocols around a common core.

The auction service required four different proxies, namely client, Web server, application server, and database proxies. Each proxy typically implements three modules: a membership protocol, a service interface, and a messaging core. The membership protocol is necessary to guarantee dynamic discovery of nodes in the validation slice. The service interface is necessary for correct communication with interacting components. The common messaging core takes shunted or logged requests to load the masked components and responds to requests made by the masked components.

Regarding state management, we currently focus solely on the soft state stored (in main memory) by application servers, namely the auctions of interest to users. To handle this state, we extend the application servers to implement an explicit state checkpointing and initialization API. This API is invoked by the proxies to initialize the state of the masked application server with that of an equivalent live component.

Our experience indicates that the effort involved in implementing proxies is small and the core components are easily adaptable across services. Except for the messaging core, which is common across all proxies, the proxies for the auction service were derived by adding/modifying 232, 307, and 274 non-comment source lines (NCSL) to the Rice client emulator [21], the Apache Web server, and the Tomcat application server, respectively. The NCSL of the application server also includes the code to implement the state management API. The MySQL database proxy was written from scratch and required only 384 NCSL.

Comparator functions. Our current set of comparator functions includes a throughput-based function, a flow-based function, and a set of component-level data matching functions. The throughput-based function validates that the average throughput of the masked component is within the threshold of an expected value. The flow-based function ensures that requests and replies indeed flow across inter-component connections that are expected to be active. Finally, the data matching functions match the actual contents of requests and replies. Due to space limitations, we only consider this last type of comparator function in this paper. We describe below how we handle the cases where exact matches are not possible because of non-determinism.

Non-determinism. Non-determinism can take several forms: in the timing of responses, in the routing of requests, and in the actual content of responses. We found that timing and content non-determinism were not significant problems in the applications we studied. On

the other hand, because data and components are replicated, we found significant non-determinism in the routing of requests. For example, in the auction service, a Web server can forward the first request of a session to any of the application servers. In PRESS, routing nondeterminism can lead to a local memory access, sending the request to a remote node, or to the local disk. The proxies need to detect that these behaviors are equivalent and possibly generate the appropriate replies. Fortunately, implementing this detection and generation was quite simple for both our services.

4.5 Example validation scenario

To see how the above pieces fit together, consider the example of an operator who needs to upgrade the operating system of a Web server node in the auction service (Figure 3). The operator would first instruct Mendosus to remove the node from the online slice, effectively taking it offline for the upgrade. Once the upgrade is done and the node has been tested offline, e.g. it boots and starts the Web server correctly, the operator would instruct Mendosus to migrate the node to the validation slice. Next, the validation harness would automatically start an application server proxy to which the Web server can connect. Once both components have been started, they will discover each other through a multicast-based discovery protocol and interconnect to form a virtual service. The harness would also start a client proxy to load the virtual service. Under trace-based validation, the harness would then replay the trace, with the client proxy generating the logged requests and accepting the corresponding replies (shown as A in Figure 3), and the application server proxy accepting requests for dynamic content from the Web server and generating the appropriate replies from logged data (shown as B). The harness would also compare all messages from the Web server to the application server and client proxies against the logged data. Once the trace completes without encountering a comparison mismatch, the harness would declare the Web server node validated. Finally, the operator would place the node back into the live service by restarting it and instructing Mendosus to migrate it to the online slice *without further changing any of its configurations*.

4.6 Discussion

Having described validation in detail, we now discuss the generality and limitations of our prototype, and some remaining open issues. We also compare our approach to offline testing and undo.

Generality. While our implementations have been done only in the context of two systems, the auction service and the PRESS server, we believe that much of our infrastructure is quite general and reusable. First, the auction service is implemented by three widely used servers: Apache, Tomcat, and MySQL. Thus, the proxies and shunts that we have implemented for the auction service should be directly reusable in any service built around these servers. Even for services built around different components, our shunts should be reusable as they were implemented based on standard communication libraries. Further, as already mentioned, implementing the proxies requires relatively little effort given a core logging/forwarding and replay infrastructure. Finally, our experience with PRESS suggests that event-based servers are quite amenable to our validation approach, as all interactions between components pass through a common queuing framework.

Perhaps the most application-specific parts of our validation approach are the comparator functions and the state management API. Generic comparator functions that check characteristics such as throughput should be reusable. However, comparator functions that depend on the semantics of the applications are unavoidably application-specific and so will likely have to be tailored to each specific application. The state management API is often provided by off-the-shelf stateful servers; when those servers do not provide the API, it has to be implemented from scratch as we did for Tomcat.

Limitations. The behavior of a component cannot be validated against a known correct instance or trace when the operator actions properly change the component's behavior. For example, changes to the content being served by a Web server correctly leads to changes in its responses to client requests. However, validation is still applicable, as the validation harness can check for problems such as missing content, unexpected structure, etc. In addition, once an instance has been validated in this manner, it can be used as a reference point for validating additional instances of the same component type. Although this approach introduces scope for mistakes, we view this as an unavoidable bootstrapping issue.

Another action that can lead to mistakes is the restart of components in the live service after validation, a step that is typically necessary to ensure that the validated components will join the live service with the proper state. However, the potential for mistakes can be minimized by scripting the restart.

Open issues. We leave the questions of what exactly should be validated, the degree of validation, and for how long as open issues. In terms of trace-based validation, there are also the issues of when traces should be gathered, how often they should be gathered, and how long they should be. All of these issues boil down to policy decisions that involve trade-offs between the probabil-

ity of catching mistakes vs. the cost of having resources under validation rather than online. In our live operator experiments with validation, we leave these decisions to the discretion of the operator. In the future, we plan to address these issues in more detail.

One interesting direction is to study strategies for dynamically determining how long validation should take based on the intensity of the offered load. During periods of heavy load, validation may retain resources that could be used more productively by the live service.

We also plan to explore a richer space of comparator functions. For example, weaker forms of comparison, such as using statistical sampling, may greatly improve performance while retaining the benefit of validation.

Comparison against offline testing. Sites have been using *offline testing* for many years [3]. The offline testing environment typically resembles the live service closely, allowing operators to act on components without exposing any mistakes or intermediate states to users. Once the components appear to be working correctly, they can be moved into the live service.

The critical difference between our validation approach and offline testing is the fact that our validation environment is an extension, rather than a replica, of the live service. Thus, misconfiguration mistakes can occur in offline testing when the software or hardware configurations have to be changed during the moving of the components to the live service. For example, adding an application server requires modifying the configuration file of all the Web servers. Although a misconfigured Web server in the offline environment can be detected using offline testing, failing to correctly modify the live configuration file would result in an error. Furthermore, other mistakes could be made during these actions and consequently be exposed to the end users.

In order to gauge the ability of offline testing to catch the mistakes that we have observed (Section 3), we assume that trivial mistakes that do not involve inter-component configuration are unlikely to be repeated in the live system. Under this assumption, offline testing would have allowed the operator to catch (1) all instances of the "start of wrong software version" category, (2) the instance of local misconfiguration that caused the database security vulnerability (assuming that the operators would explicitly test that case), and (3) some instances of global misconfiguration, such as the one in which the incorrect port was specified in the Tomcat configuration file. Overall, we find that offline testing would only have been able to deal with 17 out of the 42 mistakes that we observed, i.e. 40% of the mistakes, while our validation approach would have caught 66%.

Comparison against undo. Undo [6] is essentially orthogonal to our validation approach. Undo focuses on enabling operators to fix their mistakes by bringing the service back to a correct state. The focus of validation is to hide operator actions from the live service until they have been validated in the validation environment. As such, one technique can benefit from the other. Undo can benefit from validation to avoid exposing operator mistakes to the live service, and thus the clients, whereas validation can benefit from undo to help correct operator mistakes in the validation environment.

Assuming that undo would be used alone, all mistakes we observed would have been immediately exposed to clients (either as an explicit error reply or as degraded server performance), except for the ones that caused latent errors and vulnerabilities. Nevertheless, undo would be useful in restoring the system to a consistent state after a malicious attack resulting from the database security vulnerability problem. If combined with offline testing, undo would have helped fix the mistakes detected offline.

5 Experimental Validation Results

In this section we first describe the performance impact of our validation infrastructure on the live service using micro-benchmarks. We then concretely evaluate our validation approach using several methods.

5.1 Shunting overheads

We measured the shunting overhead in terms of CPU usage, disk, and network bandwidth for interception, logging, and forwarding of inter-component interactions in the auction service. Note that while the comparator functions do not run on the online slice, the amount of information that we have to log or forward depends on the nature of the comparator functions we wish to use for validation. Thus, we investigate the performance impact of several levels of data collection, including collecting complete requests and replies, collecting only summaries, and sampling.

Figure 4 shows percentage CPU utilization for a live Web server in the auction service, as a function of the offered load. The utilization vs. load curve for the unmodified server is labeled *base*. The curves for complete logging (used for trace-based validation) and forwarding (used for replica-based validation) are labeled *trace-val* and *replica-val*, respectively. These curves show the performance of shunting all requests and replies from both the Web and application servers. We can observe that complete logging causes an additional 24-32% CPU utilization for the throughput range we consider, whereas forwarding adds 29-39%.

A straightforward approach to reducing these overheads is to use only a summary of the responses. The *trace-summary-val* and *replica-summary-val* curves give

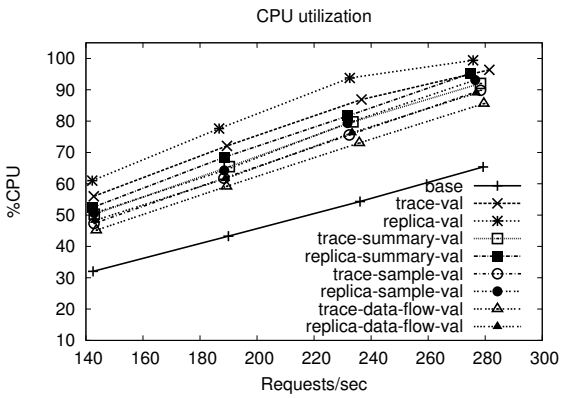

Figure 4: *Processor overhead incurred in performing various validation operations on Web server.*

the utilization for logging and forwarding, respectively, when we use the first 64 bytes of the HTTP responses. The additional CPU utilizations in this case for logging and forwarding are 18-25% and 20-27%, respectively.

A second approach to reducing the overheads is to sample, instead of collecting all requests and responses. To measure the impact of this approach, we programmed the shunts to log or forward only 50% of the client sessions, leading to the *trace-sample-val* and *replica-sample-val* curves. (This optimization was carefully implemented to avoid skewing the states of the compared components in replica-based validation.) The optimization reduces the overheads of logging and forwarding to 15-21% and 18-25%, respectively. Another sampling approach is to only shunt and compare the final input and outputs, ignoring internal messages. The *trace-data-flow-val* and *replica-data-flow-val* versions only sample HTTP requests and responses and ignore the JK messages. This approach leads to a CPU overhead of 13-19% and 16-22% for logging and forwarding, respectively.

We also examined the impact of shunting on disk and network bandwidth. We find that in the worst case, a bandwidth load of about 4 MB/s was generated. Using the every-other-session sampling method reduced the bandwidth load to about 2.5 MB/s, and the final-result-only sampling method further reduced it to 1.5 MB/s. These bandwidth results are encouraging, as they show that validation is unlikely to have an impact on throughput when using Gigabit networks and storage systems.

Overall, we find the CPU overheads to be significant for our base prototype, in many cases increasing utilization by 24%-39%, while the additional network and disk traffic was not significant. With different optimizations, we can reduce the CPU overheads to 13%-22%. These results are positive, given that *our approach loads only one or at most a few of the live components simultaneously, and only during validation*. Furthermore, since many services run at fairly low CPU utilization (e.g., 50%-60%) to be able to deal with load spikes, this overhead should not affect throughputs in practice.

5.2 Buffering overheads

State checkpointing and initialization are performed by the shunts and proxies involved in the validation of a stateful server. We make these operations atomic by first draining all requests currently being processed by the components involved in the validation. After those requests complete, we start the required state operations. During the draining and the processing of the state operations, we buffer all requests arriving at the affected components. How long we need to buffer requests determines the delay imposed on (a fraction of) the requests and the buffer space overheads.

While the delays and space overheads can vary depending on size of the state and the maximum duration of an outstanding request, we find them to be quite tolerable for the validation of an application server in our auction service. In particular, we find that replica-based validation causes the longest buffering duration. However, even this duration was always shorter than 1 second, translating into a required buffer capacity of less than 150 requests for a heavily loaded replica server.

Since the average state size is small (less than 512 bytes) in the auction service, we synthetically increased the size of each session object up to 64 KBytes to study the impact of large states. This resulted in an overall response time of less than 5 seconds, which though not insignificant, is still manageable by the validation slice.

5.3 Operator mistake experiments

We used three different experimentation techniques to test the efficacy of our validation techniques. The experiments span a range of realism and repeatability. Our live-operator experiments are the most realistic, but are the least repeatable. For more repeatable experiments, we used operator emulation and mistake injection. For all experiments, the set up was identical to that described in Section 3, but we added two nodes to implement our validation infrastructure.

Live-operator experiments. For these experiments, the operator was instructed to perform a task using the following steps. First, the component that must be touched by the operator is identified and taken offline. Second, the required actions are performed. Next, the operator can use the validation slice to validate the component. The operator is allowed to choose the duration of the validation run. Finally, the operator must migrate the component to the online slice. Optionally, the operator can

place the component online without validation, if he/she is confident that the component is working correctly.

We ran eight experiments with live operators: three application server addition tasks (Section 3.3), three server upgrade tasks (Section 3.5), and one each of Web server misconfiguration (Section 3.6) and application server hang (see Section 3.7). Seven graduate students from our department acted as operators, none of whom had run the corresponding experiment without validation before.

We observed a total of nine operator mistakes in five of the experiments and validation was successful in catching six of them. Two of the mistakes not caught by validation were latent errors, whereas the other mistake, which led to an empty `htdocs` directory, was not caught only because our implementation currently does not test the static files served by the Web servers (as already mentioned in Section 4.4). Addressing this latter mistake merely requires an extension of our prototype to process requests to static files and their corresponding responses.

Interestingly, during one run of the Web server upgrade task, the operator started the new Apache without modifying the main configuration file, instead using the default one. Validation caught the mistake and prevented the unconfigured Apache from exposure. However, the operator tried to configure the upgraded Apache for 35 minutes; after a number of unsuccessful validations, he gave up. This example shows that another important area for future research is extending the validation infrastructure to help the operator more easily find the cause of an unsuccessful validation.

Operator-emulation experiments. In these experiments, a command trace from a previous run of an operator task is replayed using shell scripts to emulate the operator's actions. The motivation for this approach is that collection and reuse of operator's actions provides a repeatable testbed for techniques that deal with operator mistakes. This approach, however, has the limitation that once the operator's mistake is caught, subsequent recovery actions in the scripts are undefined. Nevertheless, we find the ability to repeat experiments extremely useful.

The traces were derived manually from the logs collected during the operator experiments described in Section 3. In the emulation scripts, an emulation step consists of a combination or summary of steps from the actual run with the goal of preserving the operator actions that impact the system. For example, if the operator performed a set of read-only diagnostic steps and subsequently modified a file, then the trace script will only perform the file modification.

We derived a total of 40 scripts from the 42 operator mistakes we observed; 2 mistakes were not reproducible due to infrastructure limitations. Table 2 summarizes our findings in terms of coverage, i.e., mistakes caught with respect to all mistakes. Validation was able to catch 26 of the 40 reproducible mistakes; 22 of these mistakes had an immediate impact while 4 caused latent errors. Both trace and replica-based validation caught all 22 mistakes causing an immediate impact. However, single-component validation failed to catch the latent errors during the addition of a new application server. These mistakes resulted in the Web servers not being updated correctly to include the new application server. These mistakes were caught using the multi-component validation approach described in Section 4.3.

Technique (40 total)	**Coverage Impact**	
	Immediate (29 total)	**Latent** (11 total)
Trace-based	22	0
Replica-based	22	0
Multi-component	22	4

Table 2: *Coverage results of the emulation experiments.*

Validation would have caught the 2 non-reproducible mistakes as well. These mistakes had an immediate impact similar to a number of the 22 reproducible mistakes caught by single-component validation. Assuming that these 2 mistakes would have been caught, our validation approach would detect a total of 28 out of the 42 mistakes (66%) we have observed.

Mistake-injection experiments. We hand-picked some additional mistakes and injected them to test the effectiveness of our validation system. Our goal is to see if our validation technique can cover mistakes that were not observed in the live-operator experiments.

To emulate mistakes in content management, we extended Mendosus to inject permission errors, missing files, and file-corruption errors. In PRESS, injection of permission and missing files errors were readily detected by our validation infrastructure. However, some file corruption errors were not caught because of thresholds in the comparator functions; typically a fraction of the bytes of a Web page are allowed to be different. While it is necessary to allow some slack in the comparator to prevent excessive false positives, this case illustrates that the comparator functions must be carefully designed to balance the false positive rate with exposing mistakes.

We also used Mendosus to perform manipulations of configuration parameters that only impacted the performance of the component. Specifically, we altered the in-memory cache size for PRESS and the maximum number of clients for Apache in the auction service. Both mistakes resulted in the component's performance dropping below the threshold of a throughput comparator and so were caught by validation. Once again, these experiments highlight the importance of designing suitable comparators and workloads.

6 Conclusions

In this paper, we collected and analyzed extensive data about operator actions and mistakes. From a total of 43 experiments with human operators and a three-tier auction service, we found 42 operator mistakes, the most common of which were software misconfiguration (24 mistakes) and incorrect software restarts (14 mistakes). A large number of mistakes (19) immediately degraded the service throughput.

Based on these results, we proposed that services should validate operator actions in a virtual environment before they are made visible to the rest of the system (and users). We designed and implemented a prototype of such a validation system. Our evaluation showed that the prototype imposes an acceptable performance overhead during validation. The results also showed that our prototype can detect 66% of the operator mistakes we observed.

Acknowledgments. We would like to thank our volunteer operators, especially the Ask Jeeves programmers and the DCS LCSR staff members, who donated considerable time for this effort. We would also like to thank Christine Hung, Neeraj Krishnan, and Brian Russell for their help in building part of the infrastructure used in our operator experiments. Last but not least, we would like to thank our shepherd, Margo Seltzer, for her extensive comments and support of our work.

References

[1] 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 (HotOS IX)*, May 2003.

[2] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan. Magpie:Real-Time Modelling and Performance-Aware Systems. In *Proceedings of the 9th Workshop on Hot Topics in Operating Systems (HotOS IX)*, May 2003.

[3] R. Barrett, P. P. Maglio, E. Kandogan, and J. Bailey. Usable Autonomic Computing Systems: the Administrator's Perspective. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, May 2004.

[4] C. Boyapati, B. Liskov, L. Shrira, C.-H. Moh, and S. Richman. Lazy Modular Upgrades in Persistent Object Stores. In *Proceedings of the 18th ACM SIGPLAN Conference on Object-oriented Programming, Languages, Systems and Applications (OOPSLA'03)*, Oct. 2003.

[5] A. Brown. *A Recovery-oriented Approach to Dependable Services: Repairing Past Errors with System-wide Undo*. PhD thesis, Computer Science Division-University of California, Berkeley, 2003.

[6] A. B. Brown and D. A. Patterson. Undo for Operators: Building an Undoable E-mail Store. In *Proceedings of the 2003 USENIX Annual Technical Conference*, June 2003.

[7] E. V. Carrera and R. Bianchini. Efficiency vs. Portability in Cluster-Based Network Servers. In *Proceedings of the 8th Symposium on Principles and Practice of Parallel Programming (PPoPP)*, June 2001.

[8] M. Castro and B. Liskov. Proactive Recovery in a Byzantine-Fault-Tolerant System. In *Proceedings of the 4th USENIX Symposium on Operating Systems Design and Implementation (OSDI'00)*, Oct. 2000.

[9] M. Castro and B. Liskov. BASE: Using Abstraction to Improve Fault Tolerance. In *Proceedings of the 18th ACM Symposium on Operating Systems Principles (SOSP'01)*, Oct. 2001.

[10] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-Based Failure and Evolution Management. In *Proceedings of the International Symposium on Networked Systems Design and Implementation (NSDI'04)*, Mar. 2004.

[11] J. Gray. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems*, Jan. 1986.

[12] J. Gray. Dependability in the Internet Era. Keynote presentation at the 2nd HDCC Workshop, May 2001.

[13] Z. T. Kalbarczyk, R. K. Iyer, S. Bagchi, and K. Whisnant. Chameleon: A Software Infrastructure for Adaptive Fault Tolerance. *IEEE Transactions on Parallel and Distributed Systems*, 10(6), 1999.

[14] E. Kiciman and Y.-M. Wang. Discovering Correctness Constraints for Self-Management of System Configuration. In *Proceedings of the 1st International Conference on Autonomic Computing (ICAC'04)*, May 2004.

[15] X. Li, R. P. Martin, K. Nagaraja, T. D. Nguyen, and B. Zhang. Mendosus: SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of 1st Workshop on Novel Uses of System Area Networks(SAN-1)*, Jan. 2002.

[16] D. E. Lowell, Y. Saito, and E. J. Samberg. Devirtualizable Virtual Machines - Enabling General, Single-Node, Online Maintenance. In *Proceedings of the 11th International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS IX)*, Oct. 2004.

[17] B. Murphy and B. Levidow. Windows 2000 Dependability. Technical Report MSR-TR-2000-56, Microsoft Research, June 2000.

[18] K. Nagaraja, F. Oliveira, R. Bianchini, R. P. Martin, and T. D. Nguyen. Understanding and Dealing with Operator Mistakes in Internet Services. Technical Report DCS-TR-555, Department of Computer Science, Rutgers University, May 2004.

[19] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do Internet Services Fail, and What Can Be Done About It. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems (USITS'03)*, Mar. 2003.

[20] D. A. Patterson, A. Brown, P. Broadwell, G. Candea, M. Chen, J. Cutler, P. Enriquez, A. Fox, E. Kiciman, M. Merzbacher, D. Oppenheimer, N. Sastry, W. Tetzlaff, J. Traupman, and N. Treuhaft. Recovery-Oriented Computing (ROC): Motivation, Definition, Techniques, and Case Studies. Technical Report UCB//CSD-02-1175, University of California, Berkeley, Mar. 2002.

[21] Rice University. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.

[22] M. Welsh, D. E. Culler, and E. A. Brewer. SEDA: An Architecture for Well-Conditioned, Scalable Internet Services. In *Proceedings of the ACM Symposium on Operating Systems Principles (SOSP)*, Oct. 2001.