Barricade: Defending Systems Against Operator Mistakes *

Fábio Oliveira Andrew Tjang Ricardo Bianchini Richard P. Martin Thu D. Nguyen

Department of Computer Science, Rutgers University, Piscataway, NJ 08854, USA

{fabiool,atjang,ricardob,rmartin,tdnguyen}@cs.rutgers.edu

Abstract

In this paper, we propose a management framework for protecting large computer systems against operator mistakes. By detecting and confining mistakes to isolated portions of the managed system, our framework facilitates correct operation even by inexperienced operators. We built a prototype management system called Barricade based on our framework. We evaluate Barricade by deploying it for two different systems, a prototype Internet service and an enterprise computer infrastructure, and conducting experiments with 20 volunteer operators. Our results are very promising. For example, we show that Barricade can detect and contain 39 out of the 43 mistakes that we observed in 49 live operator experiments performed with our Internet service.

Categories and Subject Descriptors K.6 [*Management of computing and information systems*]: System management

General Terms Design, experimentation, management

Keywords Manageability, operator mistakes

1. Introduction

The complexity of today's enterprise computing systems poses a major challenge to system administrators.[1] Current systems comprise a multitude of inter-related software components running on many machines. To make matters worse, as computers permeate all aspects of our lives, higher demands are placed on the availability and correct operation of these systems. Many studies have shown that human mistakes are an important source of unavailability and incorrect behavior [6, 12, 14]. When mistakes are made, repairing them can be time-consuming [14]. Because mistakes can be so costly, enterprises need to hire many expert operators, who are constantly in high demand. A recent IDC study [7] shows that labor costs accounted for roughly 60% of the spending on server systems in 2006, far outweighing the spending on new servers, and power and cooling. Even worse, the same study shows that labor costs have recently been increasing at around 10% per year.

Responses from a survey of experienced network administrators and DBAs (Database Administrators) [2, 12] further substantiate this problem. Among the most common system failures reported by network administrators, 43% of them can be attributed to operator actions. Also, according to the DBAs, mistakes are responsible for roughly 80% of the administration problems they reported, corroborating an early study that illustrates the extent of DBA mistakes [6]. Finally, for three commercial Internet services, such as Google or Ebay, Oppenheimer et al. have shown that operator mistakes are a dominating cause of unavailability [14].

The characteristics of the mistakes are environment-specific, but across many environments, common mistakes include software misconfigurations and improper deployment of new or upgraded software. Another key problem is that mistakes made on one or a few machines are propagated to many others via scripts, increasing the scope of the problem and the subsequent repair time.

These observations suggest the need for mistake-free systems management, hopefully by fewer and less experienced (i.e., less costly) operators. Along these lines, previous works have proposed (1) to automate certain operator tasks [9, 19, 22], (2) to guide operator tasks that need to be performed manually [2, 3], (3) to audit changes to the system's persistent state [21], and (4) to isolate and validate operator actions before they become visible to the users or the rest of the system [11, 12]. A related approach is to undo the operator's actions when a mistake is detected [4].

Although useful in many cases, these approaches have limitations. Automation can only be applied to repetitive tasks that can be easily parameterized; moreover, a buggy automated process can rapidly spread mistakes throughout the system. Guidance, auditing, and validation require the operator to follow recommendations and/or abide by rules of behavior. Validation does not protect against actions that are performed directly on the production system, as it requires the operator to act on machines taken off-line. Finally, audit-

* This research was partially supported by NSF grant CNS-0916878.

[1] We use the terms *administrator* and *operator* interchangeably.

EuroSys'10, April 13–16, 2010, Paris, France.

ing and undo systems do not prevent mistakes from occurring in the first place and possibly spreading across the system. For example, it is impossible to undo operator mistakes that have side effects, such as sending messages to users.

In this paper, we advocate a radically different approach to dealing with operator mistakes, called *mistake-aware systems management*, that can be easily combined with the previous approaches and has none of the limitations listed above. In a mistake-aware system, an omnipresent management infrastructure defends the system against mistakes. To accomplish this goal, the infrastructure constantly monitors the operator actions and the system state to decide whether or not it should react to what the operator is doing. The reaction can prevent an observed (or potential) mistake from compromising the system (and possibly requiring a lengthy repair process). For instance, while the operator is editing a critical configuration file on a Web server, the infrastructure may decide to preventively make the same file immutable on all other Web server replicas (called a *blocking action*), until it has verified that the edits are valid. At that point, the infrastructure would re-enable writes to the blocked configuration files (called a *lifting action*). Interestingly, the blocks are often erected and lifted before they are perceived by the operator. When this is not the case, the operator can query the infrastructure to understand the reason for the blocks. Bypassing the blocks manually is possible, but only with permission from an identified *super-operator*. Mistake-aware management is most useful to prevent mistakes of less-experienced operators, but it is effective for experienced ones as well.

We propose a general framework for building mistake-aware systems and build Barricade, a prototype management system based on the framework. As our main case study, we apply Barricade to protect a multi-tier Internet service and perform 49 experiments with 18 volunteer operators, lasting more than 45 hours. Further, we analyze 43 trace-based experiments, representing more than 67 hours of interactions with the system. Our results are very promising. For example, we show that Barricade was able to detect and contain 81 out of 85 observed mistakes. Importantly, our interviews with the volunteers suggest that they do not find Barricade to be intrusive; quite the opposite, they frequently resorted to Barricade to guide their actions.

2. Framework for Mistake-Aware Management

We frame our idea of mistake-aware management in the context of large computer systems comprising interrelated and replicated components running on multiple nodes. In such an environment, the typical procedure for carrying out an operation task, e.g., upgrading the software of a replicated component, involves many steps. First, the operator (or a script) takes one or more nodes hosting the component's replicas off-line (i.e., out of active service). Second, she performs the task on these nodes, either manually or using a script, and tests if its outcome is correct. Third, she derives a procedure for performing the task on the remaining nodes, possibly scripting it. Fourth, she tests the procedure. Finally, she applies the procedure to all nodes hosting the component's replicas to complete the task.

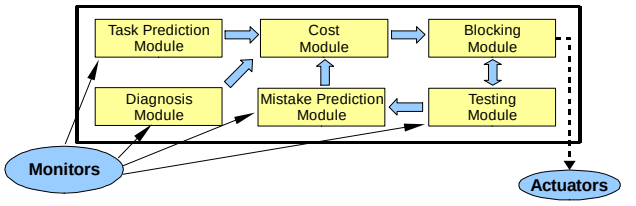

Figure 1. A general framework for mistake-aware systems.

Unfortunately, (1) the operator may take the wrong nodes off-line, (2) she may not have adequately tested her changes to the off-line nodes, (3) she may make a mistake in scripting the dissemination procedure, and (4) the testbed may not be an exact mirror image of the online nodes [12]. All of these can either harm the live system or lead to a dissemination script that contains mistakes; when this script is applied, these mistakes will be spread to the entire system.

This situation calls for a mistake-aware management system that can take the appropriate nodes off-line, recognize the task being performed, and block that task from being carried out on all replicas if the expected cost of mistakes is high. The operator actions should only be allowed to disseminate to the other nodes after they have been tested and verified to be correct by the management system.

Note that in mistake-aware management there is no need to distinguish between human operators and scripts. The approach handles scripts as though the operator were executing each step by hand. For this reason, we do not differentiate them hereafter.

To accomplish its goals, a mistake-aware management system requires that nodes to be operated upon should never be disconnected from the management system, even when the operator has taken them off-line. This allows our management system to (1) monitor the operator actions, (2) predict the task that the operator is attempting to perform, (3) identify a localized *site of operation*, a subset of nodes within which the operator may be confined with respect to the task being performed, and (4) decide whether to erect blocks to limit the operator's action to the site of operation.

We propose a general framework, shown in Figure 1, that provides the basis for different implementations of mistake-aware management systems. Our framework comprises six interacting modules: task prediction, diagnosis, mistake prediction, cost estimation, blocking, and testing. The task prediction module uses observations of operator actions obtained from the monitoring infrastructure and attempts to predict the operator task. Each task is associated with a site of operation. The mistake prediction module combines observations of operator actions and results from the test module to compute the probability of mistakes. The cost module

uses the outputs of the task and mistake prediction modules together with a cost model to compute the expected cost due to operator mistakes. The blocking module contains a set of blocking actions and uses the output of the cost module and internal estimates of the cost of actuating blocking actions to decide if and when blocks should be put in place or lifted. Typically, blocking actions erect a barrier to confine operator activities to the site of operation. Finally, the testing module contains a set of tests for validating the correctness of each known task. When outputs of the task prediction module converge to a high likelihood for one task (or a small set of tasks), the test module may decide to periodically run tests associated with the task. Successful (unsuccessful) tests may lead to lower (higher) mistake probabilities.

Containment vs. dissemination. So far, we have discussed the behavior of the management system when the operator is performing a task within the site of operation. We call this the *containment* phase. However, many tasks eventually require disseminating changes to replicas outside the site of operation. In our framework, the testing module verifies when the operator has completed a task within the site of operation. If all tests succeed, the system will end the containment phase and enter a *dissemination* phase, in which the operator is allowed to disseminate her changes.

Since the operator may make a mistake during dissemination—e.g., repeat the procedure incorrectly or run a buggy dissemination script—this phase must also be controlled. Depending on the expected cost of mistakes, the blocking module may only lift the blocks from a subset of nodes. The operator would then disseminate her changes to these nodes, and the management system would test the changes. If the tests run successfully, the system would allow her to disseminate changes to more nodes, eventually reaching all that need to be modified. Note that this assumes the operator or the dissemination script can communicate with the management system to learn the dissemination order. In fact, we envision an infrastructure that allows the operator to query the system to understand what it is doing and why. Our prototype includes such a guidance feature.

Diagnosis and repair. The process described above occurs when the operator is performing a scheduled-maintenance task. In essence, it is initiated by the task prediction module. An analogous process is triggered by the diagnosis module when one or more monitors (Figure 1) flag a possible system malfunction, requiring an operator to determine the problem and fix it. The diagnosis module transforms the monitors' output into a probability distribution of likely system problems. Each problem is associated with its corresponding site of operation. The distribution and the output of the mistake module (i.e., the probabilities that the operator will make a mistake in trying to fix each problem) are used by the cost module to compute the expected costs of operator mistakes. The blocking module uses the expected costs of mistakes and the output of the testing module to decide whether

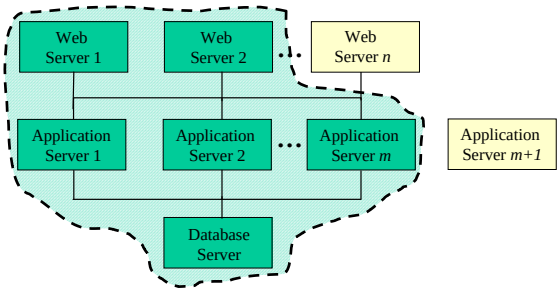

Figure 2. Barrier (dashed) erected in a 3-tier service when the operator is adding a new 2nd-tier server.

blocks should be erected or lifted as the operator tries to diagnose and repair the problem(s). When such diagnose-and-repair tasks are performed by the operator, the logical barrier created by blocking actions aims to prevent her from modifying the behavior of software and hardware components not affected by the system problem(s).

An example. Next, we describe how a management system based on our framework would behave for a specific example task: add an application server to the second tier of the 3-tier Internet service shown in Figure 2. After the task has been completed, the application server $m+1$ should be integrated into the live service. To that end, the operator needs to install, configure, and startup the application server software on the new machine. She also needs to modify the configuration of all Web servers in the first tier so that they can direct requests to the new application server.

As the operator adds the new node to the management system and starts to install and configure the application server software, the system would recognize the task via the task prediction model. The site of operation associated with this task would include the new server node and a subset of Web servers. When the task is first recognized (with high probability), the new node (application server $m+1$) would have been identified as being part of the site of operation. Any block erected at this point would limit access to the database server and other application server nodes. When the operator starts to work on a Web server node, say Web server n, that node would be added to the site of operation. Then, if a block was already up, it would be extended to limit access to the other Web servers as well, as shown by the hatched area in Figure 2. Plausible containment blocks could be: (1) for all Web servers not being operated on, prevent configuration changes and the shutdown of the Web server software; and (2) for all previously existing application servers and the database server, prevent configuration changes and the shutdown of the server software.

As the operator proceeds, the system may periodically run the tests associated with the task on Web server n and application server $m+1$. Plausible tests include checking if the application server software is running and if the Web server has been configured correctly. If the servers do not pass the tests, the containment blocks will remain in effect.

If the tests are successful, the probability that a mistake was or will be made decreases, thereby increasing the chance that the containment blocks will be lifted.

Once the containment phase is over, the operator needs to disseminate changes to the other Web servers. If the expected cost of mistakes remains high, dissemination would be a controlled process as explained above. The management system may allow the operator to modify the configuration of one additional Web server, say Web server 1. Only after such actions have been tested would the system allow her to operate on Web server 2, and so on. The same restrictions would be imposed on dissemination scripts.

Management roles. We consider 3 types of management roles: service engineers, operators, and super-operators.

Service engineers instantiate and configure the management system. They must know the architecture and topology of the target service, as well as the operator tasks.

Regular operators perform scheduled maintenance and diagnose-and-repair tasks. Our framework is designed to protect the managed service against mistakes that these operators may make as they perform their normal duties. *Our work assumes that operators are not malicious*; they simply make honest mistakes.

Super-operator has privileges to work around the management system. For example, if the system mis-predicts and blocks a regular operator from progressing with a task, the super-operator can take down the block. Moreover, complicated tasks that can significantly impact the service, e.g., a software upgrade that requires a switch over of the entire service, likely will have to be performed by the super-operator as the framework would naturally block regular operators from such actions.

In summary, service engineers configure the management system that monitors operators and may block their actions. The blocks are lifted either automatically or, in rare cases, by the super-operator.

3. Barricade

We now describe Barricade, a prototype management system based on our framework. In Barricade, all nodes in the managed (or "target") system are connected to a management server. An *actuator* and *monitors* run on each of the target system's servers. The monitors constantly observe shell commands, changes to the persistent state, and various system state attributes, and report them to the management server. As it collects information from the monitors, the management server executes models for task prediction, mistake prediction, diagnosis, and cost estimation, ordering the actuators to run tests and erect or lift blocks as appropriate.[2] For large systems, we could have multiple management servers, each one in charge of a different group of servers.

[2] We assume that the management server runs on a full-fledged machine, rather than any sort of resource-limited, embedded service processor.

In this section, we describe the parts of Barricade that are general and can be reused across different target systems. The service-specific parts, e.g., the task set and corresponding test suites, are described in Section 4.

3.1 Monitoring

The monitoring infrastructure comprises four components: a *command-shell monitor*, a *persistent-state monitor*, a *utilization monitor*, and a set of *diagnosis monitors*. The first is a modified version of the bash shell that informs the management server of the execution of relevant commands. A relevant command is any command that the service engineer considers important for task and mistake prediction. Before executing any relevant command (or commands that operate on any relevant file, as described below), the Barricade shell sends an *evidence message* to the management server reporting the imminent command execution. Every new shell process started by the operator, directly or indirectly, will register itself with the management server and forward evidence messages. In addition, *all commands executed from shell scripts interpreted by the Barricade shell are monitored in the same way as any command issued from the command line.* We currently support bash scripts.

The persistent-state monitor running on each server also relies on a list of relevant file/directory names. This monitor sends evidence messages to the management server after any such file/directory is created, deleted, or modified.

The utilization monitor sends the management server a time series of dynamic properties such as CPU and memory utilization. The time series can be used by test procedures.

Finally, diagnosis monitors (Section 3.9) inform Barricade when some service components may be misbehaving.

3.2 Actuation

The actuator carries out blocking and lifting actions. The blocking actions currently supported are *command blocking*, which prevents the execution of a command, and *file blocking*, which makes a file (or directory) immutable. Obviously, the antithetical lifting actions are also supported.

The actuator running on each server interacts with the Barricade shell by inserting entries to and removing entries from a list of forbidden commands used for command blocking. In addition, to enforce or lift a file block, the actuator respectively turns on or off the file's immutable attribute. Note that only the super-operator is allowed to execute the command (`chattr`) that changes file attributes.

3.3 Task Description

The task prediction, mistake prediction, cost, blocking, and testing modules revolve around operator tasks. Thus, as they instantiate Barricade, service engineers need to specify a list of tasks that operators may perform. This extensible list should contain, for each task: a *checklist*, a description of blocks, and a template for the site of operation.

```
<task name="Add application server">
    <checklist>
        <step name="apachectl start"/>
        <step name="apachectl stop"/>
        <step name="startup.sh"/>
        <step name="server.xml"/>
        <step name="mcast_heartbeat_db.conf/>
        <step name="workers.properties">
            <influence>apachectl start</influence>
        </step>
    </checklist>
</task>
```

Figure 3. Checklist for "Add application server."

The purpose of a checklist is to allow the testing module to verify if the operator has finished key steps of a task. Therefore, a checklist is merely a list containing key commands that must be executed and key files that must be modified as part of the task. It does not list all commands that an operator or a script may use; for instance, general-purpose commands (e.g. 'ps' or 'ls') are not part of a checklist. Moreover, a checklist does not include all details about how to execute the commands (e.g., it may omit some or all arguments) and does not specify how to change files.

For example, let us consider the task depicted by Figure 2, assuming Apache Web servers in the first tier, Tomcat application servers in the second, and a MySQL DBMS in the third. In this environment, a script to perform the task would have to: (1) copy the Tomcat distribution from an existing application server to the corresponding location on the new server; (2) modify 2 Tomcat configuration files by replacing all occurrences of the name of the server from which Tomcat was copied with the name of the new server; (3) start Tomcat; (4) modify, in each Web server, 1 Apache configuration file to add a 4-line description of the new server containing the server name, the protocol to be used for communication, the communication port, and a load balancing parameter that determines what fraction of the second-tier load should be serviced by the new server; (5) modify, in each Web server, that same Apache configuration file to add the name of the new server to the end of a list; (6) restart Apache in all Web servers. In comparison, the checklist for the same task is much simpler than the script, as shown in Figure 3.

Note that a checklist is necessary even when a script is used, as they serve different purposes: the former checks the latter's progress. Importantly, however, a checklist can be written even when it is impossible to automate a task with a script, as checklists only contain high-level information.

Although the checklist's steps are unordered, the service engineer may optionally specify dependencies among them. For instance, in Figure 3, the "influence" tag expresses a dependency between "apachectl start" and the file "workers.properties": whenever that file is modified, the testing module will mark the step "apachectl start" as pending.

In addition to writing a checklist, the service engineer has to describe containment and dissemination blocks for each task. A file block is identified by a file/directory name, whereas a command block is identified by a command name,

```
<task name="Add application server">
    <containment-reactions>
        <component type="WEB_SERVER">
            <blockdir name="/etc/httpd"/>
            <blockcommand name="apachectl stop"/>
        </component>
        ...
    </containment-reactions>
</task>
```

Figure 4. Sample containment blocks.

possibly with arguments. Blocks associated with a task are grouped per *type* of server. In the example of Section 2 and Figure 2, the blocks for the task "add an application server" and the server type "Web server" may be described as shown in Figure 4: (1) blocking configuration changes on the Web server corresponds to blocking a file (or a directory); and (2) preventing the shutdown of the Web server software is denoted by blocking a command.

Finally, each task needs a definition of the *site of operation*: the number of servers of each type that the operator must work on during the containment phase. For instance, in the example of Section 2, the site of operation of the task "add an application server" could be defined as one application server and one Web server. The actual servers are dynamically determined based on the interactions of the operator with the system. Currently, Barricade relies on a simple heuristic. For each server type specified by the service engineer (e.g., application server or Web server) in the site of operation definition, the servers that have reported the largest number of evidence messages until Barricade decides to erect blocks will be part of the site of operation.

3.4 Task Prediction Module

The task prediction module implements a model that takes the evidence messages sent by the shell and persistent-state monitors as inputs. It produces a probability distribution representing the likelihood that the operator is currently performing each task from a pre-specified task set. Next, we describe the simplest of the two task prediction models that we have implemented, and briefly comment on the other model, which is fully described in our technical report [13].

Order-oblivious Bayesian model. This task prediction model uses *Recursive Bayesian Estimation* [10] as follows. Given:

- a set T of tasks $\{t_1, t_2, ..., t_n\}$, with one, say t_n, representing an *unknown* task,
- a set A of operator actions $\{a_1, a_2, ..., a_m\}$ (the union of the actions in the tasks' checklists),
- a prior probability distribution $\{p(t)|t \in T\}$, where $p(t)$ is the probability of task t being executed in the absence of any evidence, and
- for each task t, a set of conditional probabilities $\{p(a|t)|(a \in A) \wedge (t \in T)\}$, $\sum_{a \in A} p(a|t) = 1$, where $p(a|t)$ gives the conditional probability that action a would be observed if the operator is executing task t,

the probability that each task t is the one being executed after an action a has been observed is computed as:

$$P_s(t) = \frac{P_{s-1}(t)P(a|t)}{\sum_{t' \in T} P_{s-1}(t')P(a|t')}$$

where $P_{s-1}(t)$ is the probability that task t is being executed immediately before a was observed, with $P_0(t) = p(t)$. The denominator is a normalizing factor to ensure that $\sum_{t \in T} P(t)$ is always 1.

The prior probability distribution of each task t can be set to $p(t) = 1/|T|$ and adjusted over time as Barricade collects information about relative task frequencies. The conditional probabilities are defined as follows. We define the *signature* $Sig(t)$ for each task t as the set of actions in t's checklist independent of any dependencies specified in the checklist. (A filename in a checklist matches any action that would modify that file.) Then, for all tasks t:

$$\forall a \in (A - Sig(t)), p(a|t) = \epsilon,$$

$$\forall a \in Sig(t), p(a|t) = \frac{1 - E}{|Sig(t)|}$$

where $E = (|A| - |Sig(t)|)\epsilon$.

The above equations assign equal conditional probabilities to all actions in a signature. They also assign a small (ϵ) conditional probability to each action not in a task's signature to prevent a mistake or an irrelevant command from driving the predicted probability of that task to 0. ϵ should be chosen so that $\epsilon << p(a|t)$ for all actions a in a task t's signature.

To clarify our task prediction model, consider Table 1, which shows how we instantiated the model for the case study we shall present in Section 4. The leftmost column shows the union of the actions in the checklists of three known tasks, namely t_1 ("Add an application server"), t_2 ("Upgrade the Web servers"), and t_3 ("Migrate the database server"). The last column represents t_4, a general unknown task. We chose $\epsilon = 0.01$ and assumed the same prior probabilities for all tasks, i.e., the *a priori* probability distribution is $\{0.25, 0.25, 0.25, 0.25\}$. As described above, each cell (a_i, t_j) is set to a probability larger than 0.01 or 0.01 depending on whether a_i is in t_j's signature. Thus, t_1's signature contains actions a_2, a_3, a_4, a_5, a_6, a_7, a_8, and a_{13}.

Now, suppose that the first operator action is to modify the configuration file 'workers.properties' (action a_7), which is part of the signature of t_1 and t_2. After applying the Bayesian update, the new probability distribution of tasks will be $\{0.36, 0.41, 0.03, 0.20\}$. Next, if the operator modifies the configuration file 'httpd.conf' (action a_1), which belongs to t_2's signature only, the updated task probability distribution would be $\{0.05, 0.75, 0.004, 0.196\}$.

It is worth emphasizing three key characteristics of our model. First, even if the task set has *hundreds* of tasks, the model can converge towards the right one, as long as the task signatures are unique and $\epsilon << p(a|t), \forall a \in A$ and $\forall t \in T$. This is the case even when there is some overlap among task signatures. For example, Table 1 shows a significant overlap between the signatures of tasks t_1 and t_2: 57% of the actions of t_2's signature also belong to t_1's. Even so, in all live and synthetic experiments we performed with our case study (see Sections 5.1 and 5.2), our task prediction model was accurate.

Operator Actions ($a_1 - a_{15}$)	t_1 *Add App*	t_2 *Upg Web*	t_3 *Mig DB*	t_4 *Unknown*
a_1: httpd.conf	0.01	**0.13**	0.01	0.0667
a_2: apachectl start	**0.116**	**0.13**	**0.095**	0.0667
a_3: server.xml	**0.116**	0.01	0.01	0.0667
a_4: mcast_heartbeat_db.conf	**0.116**	**0.13**	0.01	0.0667
a_5: startup.sh	**0.116**	0.01	**0.095**	0.0667
a_6: apachectl stop	**0.116**	**0.13**	**0.095**	0.0667
a_7: workers.properties	**0.116**	**0.13**	0.01	0.0667
a_8: shutdown.sh	**0.116**	0.01	**0.095**	0.0667
a_9: mysqldump	0.01	0.01	**0.095**	0.0667
a_{10}: my.cnf	0.01	0.01	**0.095**	0.0667
a_{11}: mysql_install_db	0.01	0.01	**0.095**	0.0667
a_{12}: mysqld_safe	0.01	0.01	**0.095**	0.0667
a_{13}: web.xml	**0.116**	0.01	**0.095**	0.0667
a_{14}: apachectl	0.01	**0.13**	0.01	0.0667
a_{15}: make	0.01	**0.13**	**0.095**	0.0667

Table 1. Task prediction parameters used in our case study of Section 4.

Second, if some tasks of the set are so similar as to render the probability distribution inconclusive, it is unnecessary to distinguish among them because the blocking actions associated with them would be the same. In this case, those tasks must be grouped as a single one. Recall that the purpose of Barricade is not task prediction; it is to guard the system against potentially harmful operator actions.

Third, the signature of a task is merely a list of representative commands and files; it is not meant to be exhaustive. Therefore, however complex a certain task might be, modeling it entails just producing a partial list of key commands and configuration files, which can be done provided that one knows what the task must accomplish. In fact, not only is the signature incomplete, but each individual action can also be incomplete. For example, all actions in Table 1 do not specify how the configuration files must be changed, neither do they provide specific arguments to complex commands, such as `mysqldump`. Also, note that t_3 is a fairly complex task (see Section 4.3) whose signature omits some steps, e.g., importing the database to the new MySQL installation.

Order-aware Bayesian model. The above model abstracts away some details that could intuitively improve prediction accuracy, including the order of action occurrences. For completeness purposes, we also define and investigate a task prediction model that does allow the expression of action ordering [13]. In a nutshell, we extend the above model in such a way that sequences of actions can be part of task signatures. The model augments the process of updating the probability distribution by adjusting it upon any order violation. This model complicates task modeling considerably and re-

quires more state at runtime. As we shall see, the order-oblivious task prediction model is extremely accurate in our extensive operator experiments. Thus, we do not discuss the order-aware model further.

3.5 Mistake Prediction Module

The mistake prediction module implements a model that receives operator actions and test results as inputs, and computes the probability that the operator will make one or more mistakes. Barricade computes the probabilities of mistakes being made, one per task in the task set, since our model uses task-specific information in its prediction. Next, we again only describe one of the two models we implemented. The other is described in our technical report [13].

Our model uses historical information together with runtime testing as follows. Given:

- a set of prior probabilities $\{p(mistake|t)|t \in T\}$, where T is the set of tasks,
- for each task t, a set of tests $\{te_1, te_2, ..., te_n\}$, and a set of weights $\{w_1, w_2, ..., w_n\}$, and
- a testing factor TF whose range is $[0, 1]$,

when an operator first starts a task t, our model predicts that the probability of her making one or more mistakes is $p(mistake|t)$. As the operator proceeds, Barricade will periodically run the set of tests associated with t. The result of each test i, denoted by $R(te_i)$, is mapped to 1 if it ends successfully, and 0 otherwise. Then, Barricade gives credit to the operator for tests that complete successfully using:

$$P(mistake|t) = p(mistake|t)(1 - \frac{\sum_{i=1}^{n} R(te_i)w_i}{\sum_{i=1}^{n} w_i} TF)$$

The weights provide a method for differentiating the importance of individual tests. Similarly, TF provides a method for relative weighing of the importance of test results and the prior probabilities. Note, also, that the updated probability of mistake $P(mistake|t)$ will never surpass the prior probability of mistake $p(mistake|t)$; in other words, the operator is not punished by steps not yet performed, but only rewarded by successfully finished ones.

A simple way to instantiate the prior mistake probability $p(mistake|t)$ for a task t is to rely on past executions of t. Specifically, if we have observed a task t being executed n times, with mistakes being made in m executions, then we define $p(mistake|t)$ to be m/n.[3] This approach dynamically updates the prior probabilities based on past history.

A possible approach to derive the weights from historical information is as follows. For each task, map the observed mistakes into a set of categories. Then, map each test to a mistake category that the test is designed to detect. Finally, assign weights to the tests proportionally to the frequency of mistakes in their categories. The intuition is that the more frequently mistakes in a category were observed, the more difficult it will be for the operator to pass the associated test(s), and thus the higher the reward should be. In Section 3.8 we show how the tests can be written.

Despite its simplicity, this prediction model also works very well, as we shall demonstrate in Section 5.

[3] If this history stretches over a long period of time, then it may be desirable to use some form of aging to account for changes in the service and/or operator behaviors over time.

3.6 Cost Module

The cost module uses the outputs of the task and mistake prediction modules to compute a current *expected cost of mistake* for each task. Specifically, it computes the expected cost of mistake for a task t as $ECM(t) = P(t) \times P(mistake|t) \times CM(t)$, where $P(t)$ is the current probability of t being the task the operator is performing, $P(mistake|t)$ is the current probability of a mistake being made during t, and $CM(t)$ is the estimated cost to the target system of one or more mistakes being made during t.

$CM(t)$ can be expressed using several metrics, including monetary loss, service capacity loss, or downtime. $CM(t)$ must be expressed in the same unit for all tasks; in our prototype, we use percentage of capacity loss. We set $CM(t)$ to the average loss of system capacity across all mistakes observed during prior executions of t.

3.7 Blocking Module

Our blocking module is based on simple expected-cost thresholds. For each task, service engineers should define a threshold $TC(t)$. Blocks should be put into place if $ECM(t)$ ever becomes greater than $TC(t)$. The blocks should be lifted if $ECM(t)$ subsequently drops below $TC(t)$. We could have implemented two thresholds (low and high) to prevent oscillation, but our experiments so far have not suggested the need for this extra sophistication.

Note that our current approach does not explicitly model the cost of blocking (and lifting) actions; this cost is indirectly factored into $TC(t)$. The reason is that the blocks we have needed so far incur relatively small overheads. Nevertheless, we can envision actions with high costs, e.g. backing up a database before allowing the operator to modify it. To account for these costs, one could organize the blocks into layers of protection, where each layer may be more expensive to put up but may also provide stronger protection. In this scenario, the blocking module would need to estimate the costs of each subset of blocks for each task. Then, as the expected cost of mistake increases beyond the cost of erecting some subset of blocks, those blocks would be put into place. As our experiments have not required them, we leave these models for future work.

3.8 Testing Module

Tests. In general, the tests for checking the correctness of a task can take any form, e.g., running previously collected traces and checking the results. Currently, we have an infrastructure for service engineers to write tests as assertions over

```
01: newApp = getServer(OPERATION, APPSERVER);
02: webGroup = getServerGrp(OPERATION, WEBSERVER);
03: allApp = getServerGrp(ALL, APPSERVER);
04: newApp.process("/scratch/jdk/bin/java -server -Xmx384m");
05: webGrp.config("/scratch/httpd/conf/workers.properties", SET,
"/root/workers/worker/balanced_workers") == allApp.hostname;
```

Figure 5. Sample pseudo-code of assertions.

the system state, including configuration and log files. We leveraged our previous work in which we modeled the correctness of online services with assertions [20]. Other works have taken similar approaches [15, 16].

Recall that the management server receives periodic information about each node under management from the monitors. These properties are collected and encapsulated in two forms: statistical objects encapsulate the time series generated by the periodical sampling of properties, such as CPU and memory utilization; configuration and log objects encapsulate information about the configuration of servers and their logs. These objects are associated with server objects and can be programmatically checked by assertions, which are Boolean expressions. In order to run an assertion against a particular server (or group), an API allows for binding server objects to actual servers of the target system.

For example, let us consider both the scenario portrayed by Figure 2 and the pseudo-code shown in Figure 5. The code in lines 01–03 binds objects to actual servers: `newApp` and `webGroup` are bound, respectively, to an application server (if any) and to all Web servers (if any) in the site of operation, whereas `allApp` is bound to all application servers of the system. The assertion in line 04 checks whether the application server software is running on the new application server by searching the given string against the list of all currently running processes; the assertion in line 05 checks whether a configuration parameter (whose XPath is given) of all Web servers in the site of operation has as its value a set of strings containing the hostnames of all application servers, including the new one. These assertions can be executed efficiently and so can be run frequently to check the operator's progress.

Module logic. The testing module has four important functions: (1) to determine when the containment phase of a task is over; (2) to determine when each dissemination round is finished; (3) to provide feedback to the mistake module; and (4) to provide guidance to the operator.

To accomplish function (1), the testing module verifies if the checklist of any task has been completed, and executes all *containment assertions* associated with each of the tasks for which containment blocks have been erected. The containment assertions run on the dynamically determined site of operation of the corresponding task. If the checklist of a task has no more pending steps and all containment assertions associated with it evaluate to true, the containment phase is over and dissemination will begin for that task.

To accomplish function (2), the testing module runs the proper dissemination assertions on the k servers allowed to undergo dissemination. It will move to the next round, i.e., to the next k servers, when all assertions evaluate to true and no steps are pending on the dissemination checklist that can be optionally specified. In performing functions (1) and (2), the testing module keeps synchronizing with the blocking module, since blocks need to be erected and lifted when switching from containment to dissemination, as well as from each dissemination round to the next.

To accomplish function (3), the testing module provides the mistake prediction module with information about when assertions are run and their results.

Finally, for function (4), Barricade allows service engineers to associate English text with each assertion. When the command `getinfo` is executed, the shell gets from the management server the texts associated with each failing assertion and presents the information to the operator.

Besides explaining the possible problems or remaining steps based on the assertions evaluating to false, `getinfo` provides more information, including the task Barricade thinks the operator is performing, or, during dissemination, the dissemination sequence that has been established and the servers currently allowed to undergo dissemination.

3.9 Diagnosis Module

Monitors. The diagnosis module is activated only when Barricade operates in diagnosis mode, which begins when any *diagnosis monitor* flags a possible malfunction. These monitors are also written as sets of assertions.

Diagnosis model. This model converts the output of the monitors into a probability distribution of possible service problems. Problems should be defined in terms of the coverage of the diagnosis monitors. For instance, we can define the service problem $SP1$ as follows: when $SP1$ occurs, monitors $m1$ and $m2$ would detect it with probability 1, and monitor $m3$ with probability 0. All problem definitions constitute the table of monitor coverage.

Our diagnosis model implementation is similar to that proposed in [8] in that it converts monitors' output into a probability distribution of possible problems using Bayesian estimation and the table of monitor coverage.

The output of the model triggers a flow of information analogous to that initiated by the task prediction model's output. Blocking actions are associated with each defined problem, and blocking decisions are based on comparing the cost of potential mistakes made in trying to fix a problem against a cost threshold. The goal of the blocks is to prevent the operator from mistakenly acting on servers or subsystems unrelated to the problem.

3.10 Barricade Networks

Our framework distinguishes between target and management networks. When blocks are erected during containment, the servers in the site of operation are automatically taken out of the target network by Barricade. They become

off-line for the target network, but are still accessible to Barricade via the management network. Barricade will re-integrate them into the target network right before the start of the dissemination phase. Similarly, servers whose dissemination blocks have been lifted will remain off-line until the next dissemination round.

4. Case Study: Internet Service

We now describe our experience using Barricade to make a prototype three-tiered Internet service mistake-aware. (We have also applied Barricade to a small enterprise system that mimics our department's common computing infrastructure. Because of space constraints, we do not describe this system here although we do briefly summarize results from our experimentation with this system at the end of Section 5.)

4.1 Managed System

As a case study, we chose the same service, software and hardware configurations, and tasks that we used in our first study of operator mistakes [11]. Keeping everything the same allowed us to leverage the operator traces we collected at the time in the evaluation of Barricade as well.

The system is an online auction service modeled after EBay [17]. It is organized into three tiers, Web, application, and database. The service cluster comprises two nodes in the first tier running the Apache Web server (version 1.3.27), five nodes running the Tomcat servlet server (version 4.1.18) in the second tier, and one node running the MySQL DBMS (version 4.12) in the third tier. All machines run the Red Hat 8.0 Linux distribution with kernel 2.4.18-14 to reproduce the same setup of our previous study [11]. The service uses a heartbeat membership protocol to dynamically reconfigure itself when nodes become unavailable.

4.2 Barricade Configuration

We considered the same scheduled maintenance tasks as in our earlier study: add a new application server, upgrade the Web servers, and migrate the DBMS to a different machine. We used our knowledge of the service [11, 20] to derive a signature, checklist, and suite of tests for each task. As shown in Figure 3 and Table 1, the checklist of each scheduled maintenance task comprises commands dealing with starting, shutting down, and configuring the relevant service components (Apache, Tomcat, and/or MySQL), as well as key configuration files. We configured Barricade to allow dissemination one server at a time.

Furthermore, the prior probabilities of mistakes and the costs of mistakes were computed based on our earlier observations [11]. The metric we use for $CM(t)$ is the expected percentage of capacity loss. We configured the blocking module to erect blocks whenever the $ECM(t)$ exceeds a threshold percentage of capacity loss (10%).

Finally, for the diagnosis module, we have implemented high-level monitors that check if each server component of the managed system is responding to requests.

4.3 Scheduled-Maintenance Tasks

Task 1: add an application server. This task requires the operator to: (1) copy the Tomcat binary distribution from any machine in the second tier to the new machine and configure it properly; and (2) reconfigure (and restart) the Web servers so they will direct client requests to the new Tomcat server.

The containment blocks for this task prevent the operator from changing the behavior of the Apache and Tomcat instances running on servers outside the site of operation. In particular, we defined blocks to prevent the operator from configuring, stopping, or restarting them in a way similar to the example given in Figure 4. In this task, dissemination involves propagating the Web server configuration changes to all remaining Web servers (those not in the site of operation); for this reason, the dissemination blocks we chose prevent the Web servers other than the dissemination target from being configured, stopped, or restarted.

We used a test suite comprising 25 assertions. These assertions check if Apache and Tomcat are running, and verify configuration parameters specifying the connectivity between Web and application servers, and between application servers and the database server.

Task 2: upgrade the Web servers. Here, the operators are required to upgrade Apache in all first-tier machines. The task entails downloading Apache's source code, compiling it, configuring it properly, and integrating it into the service.

Containment (dissemination) blocks protect both Apache versions from being configured, stopped, or restarted outside of the site of operation (dissemination target). 27 assertions check Apache processes, their versions, and connectivity between tier 1 and tier 2.

Task 3: migrate MySQL. This task requires the migration of the DBMS to another machine: (1) compile and install MySQL on the new machine; (2) bring the service down; (3) dump the database tables from the old MySQL installation and copy them to the new machine; (4) configure MySQL properly; (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.

Containment (dissemination) blocks prevent the configuration and start-up of Web and application servers outside of the site of operation (dissemination target). Stopping these servers is allowed because the service should be shut down during this task. The dissemination phase involves reconfiguring and restarting the Web servers and application servers that are outside the site of operation. A total of 14 assertions check connectivity between adjacent tiers, verify that Apache, Tomcat, and MySQL processes are running, and check MySQL's configuration.

4.4 Diagnose-and-Repair Tasks

We also studied the diagnose-and-repair tasks from our earlier paper. In each task of this type, we injected a problem into the service and asked the operator to diagnose and fix it. The blocking actions prevent the operator from modifying the behavior of the components not affected by the problem. For instance, if a diagnosis monitor complains about a Web server and Barricade decides to protect the system, the logical barrier will not allow the operator to stop or reconfigure any application server, the database server, and the healthy Web servers. Below, we describe each problem.

Problem 1: Web server misconfiguration and crash. To inject this problem, we first modify the Apache configuration file pertaining to the `jk` module — which gives Apache the ability to forward requests for dynamic content to Tomcat — in such a way that an Apache restart will cause a segmentation fault. Next, we force an Apache crash.

Problem 2: application server hang. Here, we force Tomcat to hang on 4 out of 5 second-tier machines. The affected application servers become unresponsive, causing the Web servers to stop forwarding requests to them.

Problem 3: disk fault in the DB server. This scenario portrays a disk fault. We inject periodic disk timeouts in the database server, according to an exponential inter-arrival distribution with an average rate of 0.03 occurrences/sec.

5. Evaluation

We now evaluate Barricade's ability to identify and contain mistakes in the auction service through 49 live experiments with volunteer operators, as well as 43 off-line experiments with traces collected in our previous study [11]. We also evaluate Barricade's intrusiveness by surveying and interviewing the operators after they completed the experiments. Further, we gauge the accuracy and convergence of our task prediction model through 114 trace-replay experiments.

5.1 Live Operator Experiments

We ran 49 experiments using 18 volunteers, none of whom participated in our earlier study. Each experiment required the operator to perform one of the six tasks described in Section 4. As we had done before, we briefed each operator on the service's architecture, design, and interface before each experiment using a graphical diagram. We also provided the same written instructions for each task in both studies.

For comparison, we ensured that the number of experiments with each task and the experience level of the operators that performed the task exactly matched those of our earlier study. In particular, the volunteers were 13 grad students, 3 full-time system administrators, and 2 students employed as part-time operators. One of the grad students was formerly a professional developer at a commercial Internet service. We classified them into three categories based on a pre-experiment interview. The system administrators and the former professional developer were considered *experts*, 4 graduate students and the 2 part-time staff members were considered *intermediates*, and the remaining 8 graduate students were *novices*. Table 2 summarizes this information.

Tasks	Operator Experience			Avg. Time (min)	
	Nov.	Int.	Exp.	Nagaraja [11]	Barricade
Task 1	8	4	2	60	45
Task 2	–	2	3	120	117
Task 3	4	2	2	140	94
Problem 1	3	3	2	80	48
Problem 2	2	1	1	90	61
Problem 3	–	–	4	120	14

Table 2. Experiments using the base Barricade configuration. Nov. = Novice, Int. = Intermediate, and Exp. = Expert.

During the experiments, we used a client emulator to drive the service at 200 requests/sec, which is approximately 35% of the maximum throughput and consistent with current server provisioning practices. The throughput is monitored and presented to the operator graphically, so that she can see how her actions or the problems we inject affect performance. Also, she can consult monitors that check if Apache, Tomcat, and MySQL are responding. These monitors are used by the diagnosis module.

Using Barricade's own infrastructure, we monitored every command and file change executed by the operators during the experiments. These traces are available at `vivo.cs.rutgers.edu`.

Base configuration. In 43 of the 49 live experiments, we used the basic Barricade implementation (Section 3) and its configuration for the auction service (Section 4.2).

We observed a total of 37 mistakes during the 43 experiments. Figure 6(a) summarizes the mistakes and their potential impact on the service. We grouped the mistakes into five categories: *unnecessary restart of SW component* means that the operator restarted (or tried to restart) a component that was working correctly; *start of wrong SW version* means that the operator restarted a component using the wrong executable; *incorrect restart* refers to the operator's starting a component in a way that prevented it from working properly; *global misconfiguration* signifies that the operator left inconsistencies in one or more configuration files that compromised the communication between multiple service components; and *local misconfiguration* denotes the operator's misconfiguring a component, causing it to misbehave. The X-axis distinguishes three ways in which the mistakes could have potentially affected the service: *degraded throughput or service inaccessible* means that the mistake led to partial or complete loss of service capacity; *incomplete component integration* means that the mistake led to a newly added component not being seen by other components; and *security vulnerability* refers to a security hole that could be exploited by a malicious attacker. For each of the three classes of potential impact, the chart has two stacked bars. The left bar shows the number of observed mistakes, whereas the right bar shows the number of mistakes contained by Barricade.

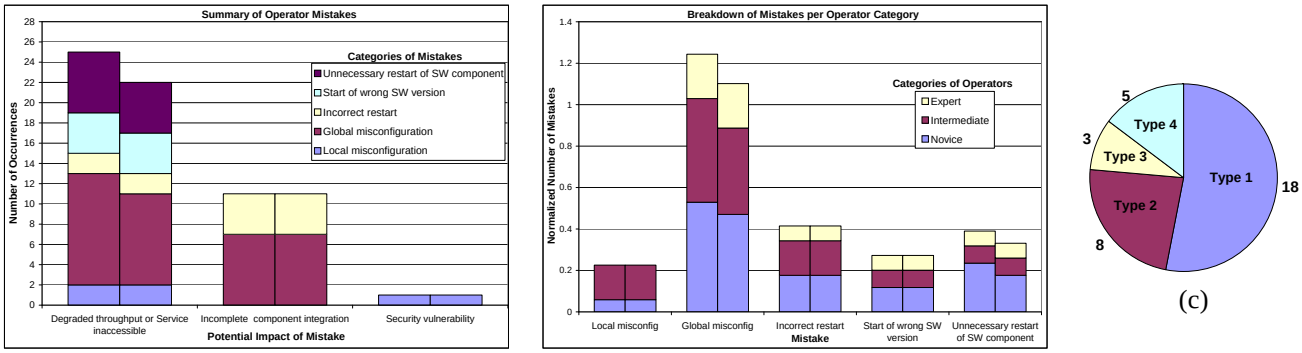

Figure 6. (a) Observed mistakes and their potential impact on the service. (b) Mistakes made by the three categories of operators. (c) Distribution of contained mistakes.

Figure 6(b) gives a breakdown of the mistakes across the experience level of the operators who made them. As no operator performed all types of experiments, we normalized the number of mistakes by dividing it by the total number of experiments performed by operators of the same category. For each mistake class, the left bar represents the total number of observed mistakes, whereas the right bar represents the number of mistakes contained by Barricade.

From Figures 6(a) and (b), we find that Barricade recognized and contained 34 out of 37 observed mistakes. Moreover, corroborating our earlier results, mistakes were made by all types of operators, even highly experienced ones.

Table 2 shows the average time spent by the operators. Interestingly, Barricade's operators spent less time on average than those of our previous study. We conjecture that this difference stems from Barricade's guidance and containment of mistakes. Nonetheless, although we rigorously reproduced our earlier testbed and followed the same methodology to categorize operators, we recognize that other factors may have contributed to this result. For instance, we used different pools of volunteers.

Contained mistakes. We identified four types of contained mistakes. Mistakes of type 1 were made in the site of operation (during the containment phase) and fixed before the operator tried to operate on other servers. Blocks were erected, but the operator did not attempt actions that would be prevented by them. Mistakes of type 2 were also made in the site of operation during containment. Differently, however, the blocks effectively stopped the operator from erroneously changing the behavior of other servers. Mistakes of type 3, on the other hand, were made during a dissemination round and were contained there. Finally, type 4 applies to diagnose-and-repair tasks and represents situations where the blocks prevented the operator from modifying the behavior of server components not affected by the problem she was trying to repair. Figure 6(c) shows the number of contained mistakes of each type.

In our experiments, mistakes of type 1 did not become type 2 because the operator realized and fixed them by resorting to guidance (*getinfo* command). Barricade's guidance also helped operators recover from mistakes of types 2 and 3, and focus on the problematic components during diagnose-and-repair tasks.

Overlooked mistakes. Barricade overlooked 3 out of 37 mistakes. Two of those were global misconfigurations: one made by a novice while adding an application server, and the other by an intermediate operator while upgrading a Web server. In the first case, a bug in the Barricade driver that parses an Apache configuration file prevented one of the test procedures from recognizing a configuration problem. The second unnoticed global misconfiguration was due to a missing test. Finally, the third overlooked mistake was an unnecessary restart of SW component made by a novice in diagnosing the Tomcat hang problem. Barricade overlooked it because the way in which the affected component was restarted was not included in the set of blocking actions. Critically, Barricade did not overlook these mistakes because of mispredictions or faulty logic; rather, it did so because of implementation bugs and incomplete testing and blocking actions provided to Barricade, all of which can easily be corrected and/or improved over time.

The role of guidance. To gauge the effectiveness of guidance, we compared our 43 experiments with those from our earlier study. Previously, 7 operators could not complete Task 1, and 3 operators incorrectly finished Task 3. This time, all operators were able to finish their tasks. Also, for Task 3 and all diagnose-and-repair tasks, the current operators were faster than those from the earlier study.

Besides the 43 experiments with Barricade's default implementation, we conducted 6 more in which we disabled *getinfo*. We observed 6 additional mistakes: 2 misconfigurations of type 2, 1 unnecessary restart, and 3 mistakes of type 4. Barricade contained 5 of them, but overlooked the unnecessary restart due to an incomplete list of blocks.

Predicted Task	Command (%)		Evidence (%)	
	3 tasks	15 tasks	3 tasks	15 tasks
Task 1	45	49	27	33
Task 2	22	28	9	16
Task 3	27	34	24	32

Table 3. Convergence of task prediction.

We conjecture that, with no guidance, when the operator makes mistakes they are more likely to be of type 2 than of type 1. The reason is that the operator might not notice a mistake as easily and might try to affect multiple system components and be stopped by blocks.

Overall, considering the 43 experiments with the default implementation and the 6 no-guidance ones, Barricade recognized and contained 39 out of 43 mistakes.

Super-operator intervention. The super-operator was needed in 3 of the 49 experiments. In two such cases, the site of operation was incorrectly detected, whereas in the other incident blocks for the wrong task were erected. These three cases were caused by bugs in our implementation, not by inaccuracy of our models. For more details on these cases and on the no-guidance experiments, please refer to [13].

Operator interviews. In order to understand how the operators felt about Barricade, we interviewed them after each experiment and asked them to fill out a questionnaire. We asked the operators who used *getinfo* to rate it, on a scale from 0 to 10, on how much it helped them (1) complete the task, (2) diagnose their mistakes, and (3) fix their mistakes. The average ratings were 7.4, 8.9, and 6.0, respectively.

Importantly, despite the fact that we configured Barricade to be aggressive about erecting barriers, Barricade did not unnecessarily disrupt the operators, except in 3 out of 49 cases that called for the super-operator. In addition, 6 operators stated that they would have liked to see more information than what was actually given by *getinfo*. Our most experienced expert operator, who was legitimately blocked once, stated that Barricade can be helpful for inexperienced operators, stopping them from making common mistakes. He added that he had made in real life the same mistake he made during Task 3 (*start of wrong SW version*), and that Barricade is a good tool to stop "cowboy operators."

Overhead. We measured the overhead imposed by Barricade on the managed servers. It adds 7% of CPU utilization on average, and takes 170 MB of RAM. Both are mainly due to the persistent-state change monitors and the machinery for running the test assertions. The network and disk overheads are negligible.

5.2 Off-Line Data Analysis

Traces of operator commands. To further evaluate Barricade, we replayed the operator traces we collected in our previous study. Of the 43 traces, 14 were mistake-free; the others had at least one mistake. In all 27 traces of scheduled-maintenance tasks, task prediction was 100% accurate, as it was in our live operator experiments. Most importantly, all 42 mistakes we recorded in our earlier study would have been contained by Barricade. In addition, Barricade would have stopped the operators from inadvertently bringing the service down by killing all Web servers.

Barricade could have misbehaved in 5 of the replayed traces. In 1 trace of Task 1, Barricade identified a wrong site of operation in exactly the same manner as the 2 cases mentioned before (due to a bug), and the super-operator would have had to intervene. Furthermore, in 4 of the 8 traces of Task 3, we considered that the super-operator could have been required, depending on how the operators would have behaved had they used Barricade. In these cases, due to a task signature overlap, blocks for the wrong task were erected in the beginning of Task 3 (during service shutdown). However, the blocks would not have prevented the operator from completing Task 3 and would have been lifted automatically as soon as the operator performed the next steps, allowing the task prediction model to converge. We observed the same scenario in one of our live experiments for Task 3 and the super-operator was not needed.

Task prediction accuracy and convergence. To gauge the accuracy and convergence of our task prediction model in a more challenging setting, we expanded the task set by generating and adding signatures for 12 more tasks, bringing the total to 15 and increasing the sum of actions of all checklists from 15 to 78. The new tasks included general management and network-related tasks, such as adding a user, changing the DNS setting, and upgrading the kernel.

In our evaluation, we used 57 traces of scheduled-maintenance tasks: the 27 we collected earlier [11], and the 30 we recorded from our recent live experiments. We replayed these traces twice: first, considering only the original 3 tasks; then, using the extended set of 15 tasks. Note that our automatic trace replayer can only receive input from the command-shell monitor. Disregarding the persistent-state change monitor made our data analysis rather conservative, as the ignored data would have considerably sped up the model convergence in all 114 cases. Convergence would have been improved because each unreported file change translated into one fewer probability distribution update.

The task was predicted correctly in all 3-task and 15-task runs, except for the one misprediction mentioned in Section 5.1 (super-operator intervention). Table 3 shows how task prediction converged to the task being replayed with more than 80% of certainty, considering the 3-task and the 15-task scenarios. The second and third columns express convergence as the percentage of the total number of commands executed, whereas the last two columns express it as the percentage of all pieces of evidence that influenced task prediction. Overall, these results show that having more tasks slows down prediction somewhat, but prediction accuracy remains close to 100%.

5.3 Case Study: Enterprise Infrastructure

Recall that we also instantiated Barricade for an enterprise system. This system comprises eight machines: two desktops, one DNS server, two mail servers, and three authentication servers. Blocks in this system prevent operator actions on replicas. To evaluate Barricade, we performed 15 experiments (5 per scheduled-maintenance task) with 6 operators, 2 of which did not participate in the other case study. Barricade contained 36 of the 39 mistakes the operators made. The super-operator was needed in 2 out of the 15 experiments. More details can be found in [13].

6. Discussion

Scope. Currently, Barricade targets systems with replicated components (at the entire node granularity). That is, Barricade seeks to contain a particular task within a small part of the overall system (the site of operation) and attempts to ensure that the task has been performed correctly before it is repeated on (disseminated to) all replicas. However, this does not mean that mistake-aware management cannot be extended to tasks that involve non-replicated components. For example, one could imagine the partitioning of a complex task into multiple phases. In this case, the mistake-aware system would try to ensure that each phase has been completed successfully before allowing the operator to proceed to the next phase. We leave this for future work since complex systems typically involve replication and so our current system provides a significant advance.

Models. The models used in our current prototype do not rely on machine learning techniques. A service engineer is expected to instantiate them explicitly. In fact, as seen in our results, service engineer's mistakes can lead to Barricade not detecting mistakes or requiring super-operator intervention. However, as already shown, *Barricade does not have to be perfect.* It only has to be sufficiently accurate to contain most mistakes while not annoying operators with false negatives.

Clearly, the task prediction and diagnosis models provide the foundation for the rest of the Barricade infrastructure. As shown in Figure 1, they are the starting points for the flow of information through the framework. The convergence of these two models determines which mistake and cost predictions must be used, what test procedures must be run, and, ultimately, what blocks must be erected. If these models are inaccurate, Barricade could potentially block parts of the system that the operator needs access to and not block parts of the system that should be protected.

Finally, note that task prediction would not be needed if the operator could be trusted to accurately state the task before beginning. However, task prediction would still be extremely useful for detecting deviations leading to mistakes.

Scalability. As the target system evolves, the service engineer may have to update task descriptions (see Section 3.3) or add new tasks to Barricade. Adding a new task consists of adding its (1) checklist, (2) test procedures, (3) list of blocks, (4) prior mistake probability, and (5) average cost of mistake.

Obviously, the service engineer must have knowledge about the task in order to produce effective checklist, test procedures, and blocking actions. However, as discussed in Section 3.3, the service engineer can ignore many details of a task when specifying its checklist, thereby minimizing the required effort. Also, while Barricade's effectiveness depends on comprehensive testing and blocking, their specifications do not have to be perfect for Barricade to be useful. Finally, the values for the task's prior mistake probability and average mistake cost may either derive from any previous executions of the task (without Barricade), or be assigned by the service engineer based on her knowledge and experience. Critically, all aspects of a task specification can be improved over time.

Operator monitoring. The current Barricade prototype relies on an instrumented `bash` shell. In a production system, the interface between the operator and Barricade will likely be richer. However, this interface must be carefully controlled as Barricade needs to accurately monitor and potentially block operator actions.

7. Related Work

Operator behavior and mistakes. Despite their critical importance, few papers have studied operator mistakes [4, 6, 11, 12, 14]. Gray [6] studied the causes of outages of Tandem systems, concluding that operator mistakes were a dominant reason. Similarly, Oppenheimer *et al.* [14] studied the causes of outages of three large-scale Internet services, reaching the same conclusion and characterizing the nature of the mistakes. We [12] previously conducted a survey with database administrators to study and characterize the problems they face, also concluding that human mistakes was a major cause of the reported problems.

Brown [4] conducted a few operator experiments with a one-node prototype email system supporting system-wide undo. In our previous work [11], we conducted extensive live operator experiments with our prototype auction service. In this work, we extend these results by performing even more experiments with both the auction service and an additional target system, adding more publicly available data on operator behavior and mistakes to this body of knowledge.

Dealing with operator mistakes. As we mentioned in the Introduction, some works have focused on dealing with mistakes [3, 4, 9, 11, 12, 19, 21, 22]. These approaches, namely automation [9, 19, 22], operator guidance [2, 3], persistent-state auditing [21], system-wide undo [4], and validation of operator actions [11, 12] have limitations. This paper proposes a novel approach that does not have the limitations of the previous approaches and can nicely be combined with them. For example, validation [11, 12], which we previously proposed to verify the correctness of operator actions in an

isolated extension of the online system, can be used to implement our framework's testing module. As another example, the recommendation system proposed by Bodik *et al.* [3], which suggests to the operators possible solutions to recurring problems, could be part of Barricade's guidance.

Operator modeling. Traditional operator modeling has focused on systems in which the cost of a mistake is enormous, possibly resulting in the loss of human life, e.g. [18]. Due to the high cost, these systems rely on complex models. Internet services and enterprise (non-mission-critical) computer systems are very different and amenable to simpler models, like the ones we used in Barricade with accurate results.

Fault containment in distributed systems. Fault containment is a widely used technique in distributed systems, e.g. [1, 5]. The containment of a node fault can be done by removing the faulty node from service and/or checking all interactions of the faulty node with other nodes. Although conceptually similar, our notion of mistake-aware management differs significantly from standard fault containment: (1) it focuses solely on containing faults resulting from the interaction of a human with the distributed system; (2) it *proactively* contains potential mistakes (i.e., faults that have not happened yet) within a *dynamically changing* site of operation; (3) the types of blocks we use for containment are completely different from those used in previous containment works; and (4) by targeting operators and being proactive, our work relies on the ability to predict tasks and mistakes.

8. Conclusions and Future Work

In this paper, we proposed a mistake-aware management framework for protecting large computer systems against operator mistakes. We have also built a prototype system and used it to transform an Internet service and an enterprise-style infrastructure into mistake-aware systems. An extensive set of experiments with volunteer operators showed that our framework and management system are capable of detecting and confining the vast majority of mistakes. Furthermore, our management system achieves these results without interfering with the operator unnecessarily, even though we configured it to aggressively block potentially harmful operator actions. On the contrary, operators frequently resorted to our system for guidance in performing their tasks. We conclude that a well-designed combination of system mechanisms, probabilistic models, and proper configuration by service engineers can create the basis for a completely different and effective approach to systems management.

We are currently studying how to adapt Barricade to handle security-related mistakes, which is more challenging since the operator may be malicious (insider attacks). We are also considering applying our approach to larger and more complex target systems. Finally, we plan to investigate learning models that will ease the effort for instantiating new mistake-aware systems and evolving existing ones.

Acknowledgements

We thank the volunteer operators, who donated a considerable amount of time to this study. We also thank the *EuroSys'10* reviewers and our shepherd, Ashvin Goel, who helped us to improve the paper.

References

[1] AZAR, Y., KUTTEN, S., AND PATT-SHAMIR, B. Distributed Error Confinement. In *Proceedings of the 22nd ACM Symposium on Principles of Distributed Computing (PODC'03)* (2003).

[2] BIANCHINI, R., MARTIN, R. P., NAGARAJA, K., NGUYEN, T., AND OLIVEIRA, F. Human-Aware Computer System Design. In *Proceedings of the 10th Workshop on Hot Topics in Operating Systems (HotOS X)* (2005).

[3] BODIK, P., FOX, A., JORDAN, M. I., PATTERSON, D., BANERJEE, A., JAGANNATHAN, R., SU, T., TENGINAKAI, S., TURNER, B., AND INGALLS, J. Advanced Tools for Operators at Amazon.com. In *Proceedings of the 1st Workshop on Hot Topics in Autonomic Computing (HotAC'06)* (2006).

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

[5] DEMIRBAS, M., ARORA, A., NOLTE, T., AND LYNCH, N. A Hierarchy-Based Fault-Local Stabilizing Algorithm for Tracking in Sensor Networks. In *Proceedings of the 8th International Conference on Principles of Distributed Systems (OPODIS'04)* (2004).

[6] GRAY, J. Why do Computers Stop and What Can Be Done About It? In *Proceedings of 5th Symposium on Reliability in Distributed Software and Database Systems (SRDS'86)* (Jan. 1986).

[7] IDC. IDC Virtualization 2.0: The Next Phase in Customer Adoption, December 2006.

[8] JOSHI, K. R., HILTUNEN, M., SANDERS, W. H., AND SCHLICHTING, R. Automatic Model-Driven Recovery in Distributed Systems. In *Proceedings of the 24th IEEE Symposium on Reliable Distributed Systems (SRDS'05)* (2005).

[9] KEPHART, J. O., AND CHESS, D. M. The Vision of Autonomic Computing. *IEEE Computer 36*, 1 (Jan. 2003).

[10] LEE, P. M. *Bayesian Statistics: An Introduction*, 3rd ed. Arnold, London, 2004.

[11] NAGARAJA, K., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. Understanding and Dealing with Operator Mistakes in Internet Services. In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation (OSDI'04)* (2004).

[12] OLIVEIRA, F., NAGARAJA, K., BACHWANI, R., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. D. Understanding and Validating Database System Administration. In *Proceedings of the USENIX Annual Technical Conference* (2006).

[13] OLIVEIRA, F., TJANG, A., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. Barricade: Defending Systems Against Operator Mistakes. Tech. Rep. 655, Dept. of Computer Science, Rutgers University, Mar. 2009.

[14] OPPENHEIMER, D., GANAPATHI, A., AND PATTERSON, D. 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)* (2003).

[15] PERL, S., AND WEIHL, W. E. Performance Assertion Checking. In *Proceedings of the 14th ACM Symposium on Operating Systems Principles (SOSP'93)* (1993).

[16] REYNOLDS, P., KILLIAN, C., WIENER, J. L., MOGUL, J. C., SHAH, M. A., AND VAHDAT, A. Pip: Detecting the Unexpected in Distributed Systems. In *Proceedings of the 3rd Symposium on Networked Systems Design and Implementation (NSDI'06)* (2006).

[17] RICE UNIVERSITY. DynaServer Project. http://www.cs.rice.edu/CS/Systems/DynaServer, 2003.

[18] SHORROCK, S. T. Errors of Perception in Air Traffic Control. *Safety Science 45*, 8 (2006), 890–904.

[19] SU, Y.-Y., ATTARIYAN, M., AND FLINN, J. AutoBash: Improving Configuration Management with Operating System Causality Analysis. In *Proceedings of the 21st ACM Symposium on Operating Systems Principles (SOSP'07)* (2007).

[20] TJANG, A., OLIVEIRA, F., BIANCHINI, R., MARTIN, R. P., AND NGUYEN, T. Model-Based Validation for Internet Services. In *Proceedings of the 28th International Symposium on Reliable Distributed Systems (SRDS'09)* (2009).

[21] VERBOWSKI, C., LEE, J., LIU, X., ROUSSEV, R., AND WANG, Y.-M. LiveOps: Systems Management as a Service. In *Proceedings of the 20th Large Installation System Administration Conference (LISA'06)* (2006).

[22] ZHENG, W., BIANCHINI, R., AND NGUYEN, T. Automatic Configuration of Internet Services. In *Proceedings of EuroSys 2007* (2007).