Operator Behavior Modeling and Analysis
Case Study: Three-tier Internet Service

Christine Hung, Neeraj Krishnan, Fábio Oliveira, Brian Russell
Department of Computer Science, Rutgers University
110 Frelinghuysen Rd, Piscataway, NJ 08854

Abstract

Operator errors are a major cause of outage in Internet Services and there is a lack of tools to assist operators in administering these services. This is because there is no model of operator behavior. In this paper, we describe an infrastructure for collecting data on how human operators administer a three-tier Internet Service and build Operator Models based on the data collected.

1 Introduction

Internet services have become popular in human activities. A plethora of complex service architectures harbor dependencies that pose a challenge to operators whose actions can cause severe side effects. Nonetheless, systems research has focused on performance, availability and scalability, whereas maintainability has received far less attention.

Operators play an important role in tending Internet services. They routinely maintain multi-tiered clusters of commodity components, each performing similar functions and contacting the next tier for further computations. Ironically, the state-of-the-art provides no tools or procedures to assist the operator, and system designs are oblivious to human error.

Our work is motivated by data gathered in "Why do Internet Services fail and what can be done about it?". In this study, real-world statistics from three different online services were collected, and the results show half of the operator errors cause service failures and operators contribute up to 75% of MTTR [10]. The Recovery-Oriented Computing group at Berkeley [11] continues this research by conducting experiments with human operators on tested systems. To start, Aaron Brown developed a learning curve metric that includes factors such as time to reach optimality, number of errors made while reaching optimality, and skill retention [5]. Next, Brown et al. created an undo mechanism for operators allowing them to rewind and repair damages caused by human error [4, 1]. Another approach Brown et al. take is to build a dependability benchmark which includes both system and operator. This benchmark measures how the operator perturbs system throughput and whether those perturbations are reactive (i.e. responding to an injected fault) or proactive (i.e. performing a system maintenance task) [3, 2]. All these experiments are carried out with a token number of operators.

To follow up on aforementioned works, we elect to create an operator model that would describe, and in some cases predict, human operator behavior based on a series of experiments with human test subjects. We choose two complementary methods to analyze our data: Represent operator actions as a Bayesian Network, and examine operator influenced system states via a Markov Decision Process. The Bayesian Network model can be used to infer probability of successfully completing an experiment, causes of failure, expected number of errors in an experiment, etc. Markov Decision Process alternatively provides optimal policies to guide an operator through a task or anticipate the operator's next move [8].

Thus far, none of the existing studies attempt to model the human-system interactions. We believe there are several advantages to having an operator model:

- System designers could incorporate our model to help the system make better decisions with regard to operator actions and errors.
- System designers could determine which system information to present to the operator and how to make that presentation beneficial to the operator.
- Operators could be more aware of their actions and make better decisions while looking after the system.

In this study, we focus on a methodology to devise an operator model with formal specifications capturing operator-system interactions.

2 Methodology

There are two stages to this study: collecting data and creating a model. We collect data by running experiments with

human subjects on a tested three-tier Internet service. We log commands operators issue to the system, the output of those commands, and the system throughput while operators perform the assigned task. Next, we analyze these data to produce an operator model. We then use Bayesian Networks and Markov Decision Processes to model operator interaction with the assigned service.

2.1 Experimental setup

At the very heart of the testbed we adopted to measure the impact of operators' actions lies a three-tier Internet service running an on-line auction application modeled after EBay. The code for such an on-line auction service — RUBiS — is publicly available and is part of the DynaServer project [6]. In order to make it suitable to our long-running experiments with operators, we had to carry out a number of modifications to the DynaServer code; chiefly, we had to alter the client emulator, extending some data structures so that our long-lasting experiments did not cause data overflow. The system exposed to the operators is comprised of 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). Before beginning a given experiment, we set up the aforementioned client emulator to generate load to the service. A number of concurrent clients repeatedly issue a request, receive and parse the response, "think" for a while and follow a link contained in the response based on a user-defined Markov model.

Another important component of our experimental setup is a monitoring infrastructure which is responsible for recording and associating a timestamp with every single command (and the corresponding result) executed by the operator, as well as for measuring the system throughput on-the-fly, presenting it to the operator so that she can visually notice whether or not her actions impact the system performance. In addition, we make use of Mendosus [7], a handy fault injection tool that allows us not only to inject various types of software and hardware faults, but also to create a virtual network environment.

2.2 Experiments

The experiments we have designed and run can be categorized as either scheduled maintenance tasks or diagnose-and-repair tasks. The former category encompasses tasks such as adding a new component to the system, migrating data to a different machine, and the like. The latter represents experiments during which we inject a fault and ask the operator to both discover what is wrong with the system and fix it. This classification of tasks differs from that suggested in a previous work [2]. Table 1 summarizes the classes of experiments we have conducted in our testbed.

We took quite a few steps before letting our volunteers loose on our auction service. First, we defined which tasks and faults of a three-tiered Internet service are of interest to us. Factors such as how much impact those tasks and faults can have on the system, how reality driven they are, and how feasible it is to turn them into carefully controlled experiments are all considered. Once we decide on a set of experiments, we produce a series of instructions with varying degrees of details and test those instructions on the authors. We run the experiments with test subjects only when we deem the tasks and faults to be sufficiently complex but ultimately achievable.

Before having the operator interact with the system, we provide her with conceptual information on the system architecture, design and interface; we convey this information verbally and give the operator a graphical representation of the system which she can refer to at any time. Afterwards, we give the operator two sets of instructions: general directions concerning the system interface and instructions specific to the task the operator will be doing. The operator is allowed to refer to both sets of instructions during the experiment.

The very first task the operators face, whose accompanying instructions are fairly detailed, gives them the opportunity to understand important system configuration issues. As suggested in a previous work [2], both this introductory task and the high-level conceptual information on the system architecture furnished to the operators help them build a mental model of the system as a whole, mitigating discrepancies in background among different operators. It is worth mentioning, however, that due to time constraints we did not run the warm-up experiment with all operators.

In addition to logging operator actions and system throughput, the authors were present during the run of an experiment. We documented our own observations and conclusions about the particular run and the test subject's cognitive process with respect to the given task. The test subjects were further interviewed, at the end of the experiment, with regard to their actions and decisions. However, these interviews were not conducted with established interview protocols and hence cannot be considered quantitatively valid.

2.3 Modeling

While our model is unlikely to be complete, given we are not conducting our experiments on a statistically valid sample, we can answer some questions about how human operators interact with a three-tier Internet service. For instance, what kind of operator errors lead to how much lost throughput? What operator actions increase throughput? What con-

Task Category	Subcategory
Scheduled Maintenance Task	Node addition
	Data migration
Diagnose-and-Repair Task	Software misconfi guration
	Application crash
	Hardware failure

Table 1: *Categories of experiments carried out on our three-tier Internet service.*

ditions are likely to lead to operator error and how frequent are those errors?

Operator action logs, system throughput graphs, and authors' on-site observations are all used in analyzing our experiment data. We build a Bayesian Network model of operator actions and also a Markov Decision Process Model acting on a matrix of operator influenced system states. The Bayesian network provides a snapshot of how a particular operator action impacts the rest of the system and where is a mistake most likely to occur. The Markov process examines a matrix of operator influenced system states and produces optimal path policies indicating what is the best action for an operator to take when the system gets into a particular state.

3 Conducted experiments

3.1 Warm up experiment

We have designed a warm-up experiment aimed at providing a means by which our operators could familiarize themselves with our testbed and with system configuration details. In the warm-up task the operators were required to add a new web server to the system. This experiment offered operators the opportunity to delve into important issues pertaining to tier-to-tier communication and helped them crystalize the information we had conveyed orally.

In our study we did not take into consideration the mistakes made by our operators during this "warm-up period".

3.2 Adding a new application server

In this experiment we ask the operator to add a new Tomcat servlet server to the second tier. 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 with the effect that the newly added Tomcat can actually receive and process requests forwarded by Apache. The experiment in question 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 twelve Computer Science graduate students and two professional programmers as operators, with an average time per run of one hour. Two of them were totally successful in the sense that they did not make any configuration mistakes, nor did they affect the system's ability to service client requests more than what was strictly necessary. On the other hand, we noticed some mistakes with varying severity among the other operators, namely:

Apache misconfigured. This was the most popular mistake by far. We recognized four different flavors of this mistake, each one impacting the system differently. In the least severe version of Apache misconfiguration, operators forgot to add the name of the new machine to the very last line of the Apache configuration file which specifies the Tomcat server names; as a result, even though Tomcat was correctly started in 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 analyzed the wrong log files and gave up searching for the correct one. Although this misconfiguration did not affect the system performance immediately, it introduced a latent error.

Another flavor of Apache misconfiguration was subtle and severe in terms of performance impact. This time, one operator modified the configuration line that was missed by other operators as mentioned in the above paragraph, but he typed an extra space before the name of the new Tomcat server. Unfortunately, the Apache parser is not tolerant and the extra space made the module responsible for forwarding requests to the second tier (`mod_jk`) crash. The outcome of the operator action was the system's inability to forward requests to the second tier and, accordingly, a 90% throughput decrease. The operator noticed the problem by looking desperately at our performance monitoring tool, but could not find the cause.

One more mistake pertaining to Apache misconfiguration was clearly a symptom of operators' not paying any attention to the task. The configuration file mentioned in the

previous two paragraphs ended up with two identical application server names. This also made `mod_jk` crash and led to a severe throughput drop: about 50% of decrease when this mistake was committed in one web server, and about 90% of decrease whenever both web servers were compromised.

In the last flavor of Apache misconfiguration we noticed, the aforementioned Apache configuration file was not modified to reflect the addition of the new application server. This error resulted in the inability of Apache to forward requests to the new application server, in much the same way as forgetting to modify the last line of such a file did.

Apache incorrectly restarted. In this case, the operators reconfigured one Apache distribution and launched the executable file from another one (There were two Apache distributions installed in the machines of our testbed). As one would expect, this error made the affected web server machine become unable to process any client requests.

Bringing down both web servers. Some operators, while reconfiguring the system to add a new application server, unnecessarily shutdown both web servers and, as a consequence, made the whole service unavailable. When asked about this mistake, some operators confessed to their missing this point, whereas other operators said that they would not have performed such a harmful action if they were dealing with a real Internet service.

Tomcat incorrectly started. Another observed mistake was the operators' inability to start Tomcat correctly. The operators forgot to obtain root privileges before initiating Tomcat. Also, in some cases, the operators started Tomcat multiple times without killing processes remaining from the previous Tomcat start up. This scenario led to Tomcat's silently dying; to make matters worse, since Tomcat's heartbeat — which is a separate process — was still running, the web servers did not take the corresponding node off-line and continued forwarding requests to a machine unable to process them. The result was obvious: degraded throughput.

3.3 Database migration

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 better disk and a faster CPU. Inasmuch as the database machine is the bottleneck of our testbed and the system is saturated when the aforementioned slow machine is used, the expected outcome of this experiment is a smoother throughput graph. The more powerful machine enables the system to keep up with the load imposed by the client emulator.

This experiment involves several steps most of which are considerably time-consuming due to the huge size of our database: 4 GB. In short, the operators are required to: (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 by modifying the `my.cnf` file; (5) initialize MySQL and create an empty database; (6) import the dumped files into the empty database in order to populate its tables; (7) modify the relevant configuration files in all application servers so that they can forward requests to the new database machine; (8) start up MySQL, all application servers and web servers.

Six Computer Science graduate students and two professional programmers performed this task; the average time per run was 2 hours and 20 minutes. We observed some operator mistakes while database migration was taking place, namely:

No password set up for MySQL root user. During MySQL configuration, the MySQL root user was not given a password. This operator mistake led to a severe security vulnerability, allowing anyone to execute any operation on the database.

User not given necessary privileges. As part of the database migration, operators are supposed to ensure that the application servers running on the second tier are able to connect to the database and issue the appropriate requests. This task requires not only reconfiguring Tomcat to forward requests to the new database machine, but it also calls for granting the proper privileges to the MySQL user which Tomcat makes use of to connect to the database. Some of our operators did not grant the necessary privileges to such a user, preventing all application servers from establishing connections to the database; as a consequence, all Tomcat threads eventually got blocked and the whole system became incapable of servicing any incoming client requests.

Apache incorrectly restarted. This is actually the same mistake we observed during the task commented previously (add application server). In the context of database migration, some operators launched the executable from the wrong Apache distribution (recall that the machines of our testbed have been provided with two Apache distributions) while restarting the whole service after completing the database installation. Once again, the service's ability to process client requests was totally ceased.

Database installed in the wrong disk. The new machine to which operators were asked to migrate MySQL had two disks: one 15K RPM SCSI and one 7200 RPM IDE. Needless to say, given that the database machine was known to be the bottleneck of our system and database migration was demanded so that the service could keep up with the load imposed by the clients, the operators should not have hesitated to install MySQL in the powerful SCSI disk. The operator who failed to do so jeopardized the service availability by underprovisioning the system.

3.4 Apache misconfiguration and crash

By asking the operators to add a new application server to the system, as previously described, we unveiled the error most frequently committed: Apache misconfiguration. In accordance with our finding that misconfiguring Apache is a very likely operator mistake, we decided to inject such a fault into the system.

In this experiment, the system starts operating normally. At some point after the experiment has begun, in one of the web server machines we modify the configuration file pertaining to the `mod_jk` module in such a way that Apache will crash due to a segmentation fault whenever an attempt at restarting it is made; then, we force Apache to crash. As soon as Apache is abnormally terminated, the throughput decreases to half of its prior value. The operators' task is to discover what is going on with the system and to fix the problem so that normal throughput can be regained.

This experiment was presented to six Computer Science graduate students and two professional programmers. Two operators were able to both legitimately understand the system's malfunctioning and fix it. However, all operators — even the successful ones throughout their interacting with the system — made some mistakes, most of which aggravated the system problem. Below we point out the relevant operator errors observed.

Misdiagnosis. Misdiagnosis was, the root cause of all other mistakes we noticed over all runs of this experiment. Due to misdiagnosis, operators unnecessarily modified configuration files all over the system which, in the worst case, caused the throughput to drop to zero. The apparent reason for such a 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.

We also noticed the popular mistake of starting the wrong Apache distribution (as previously commented), severely degrading the throughput or making it drop to zero. A few operators were willing to suggest the replacement of hardware components as a result of incorrectly diagnosing disk and memory faults.

3.5 Intermittent disk timeout on the database server

This experiment is intended to unveil the operator reactions in the face of a sudden and unknown hardware failure. In particular, by means of the Mendosus fault injection tool, we force a disk timeout to occur periodically in the disk of the machine hosting the database. The fault is injected according to an exponential inter-arrival distribution with an average rate of 0.03, which stands for 0.03 occurrences per second. Inasmuch as the database machine is the system bottleneck, the periodical disk timeouts produce a fluctuating throughput graph.

Similar to the "apache misconfiguration and crash" experiment, the system is initially under normal operation. Then, after some time, we trigger Mendosus to start injecting the disk timeouts.

Due to the time constraints we were subject to, we ran this experiment only twice. The operators who participated in this experiment — two professional programmers — were not able to discover the problem of the system after 2 hours and 30 minutes. Throughout their endeavoring, they made some mistakes worth mentioning.

Misdiagnosis. As observed in the other fault injection experiment described above, the incorrect diagnosis of the cause of the system malfunctioning has the potential of misleading the operator to perform unnecessary and, sometimes, harmful actions. During the "intermittent disk timeout" experiment, misdiagnosis influenced by error messages written to log files made the operators modify a number of configuration files. For instance, one of the operators decided to change the port which Tomcat uses to receive requests from Apache, with the effect that all application servers became unreachable.

One of the operators actually analyzed the main kernel log and saw the message logged by the disk driver reporting that the disk was not behaving properly. The operator totally ignored such a message; when interviewed, he told us that he had overlooked it because "this kind of error message appears all the time in the operating system log files".

The incorrect diagnoses made by operators during this experiment include DoS attack and network problem between the second and third tiers.

4 Bayesian Network Model

In this section we build an experiment model and an operator model. The intent is to capture operator behavior and errors while she is performing experiments. There are two classes of experiments - maintenance, and fault detection and recovery. For maintenance experiments, the operator is

provided with a set of instructions. For fault detection experiments, a fault in injected in the system and the operator has to detect and fix it. There are no instructions provided. This work is limited to modeling maintenance experiments and operator behavior while performing such experiments. We start by looking at a typical experiment, then model the experiment as a set of dependent trials, and model operator actions with a Bayesian Network. We end with a discussion of the limitations of this approach.

4.1 A Typical Experiment Run

A typical experiment consists of an operator performing a task based on a set of instructions. Each instruction states what is to be done, without always explicitly saying how to do it. There is no hard time limit on the run time of the experiment. It is not necessary that operators complete the task, although for the experiment we consider (adding an application server), all operators did go through all instruction steps. Operators usually make multiple attempts at getting each step right. They may revisit a step if some future step helps in understanding or completing it. Also, there are cyclic dependencies between some steps, so operators iteratively perform that set of steps.

4.2 An Experiment Model

We model an experiment as a set of dependent trials. A trial is an experiment with a single attempt at performing each step. Consider an experiment with three steps. Say the operator completes the first step correctly in a single attempt. She then fails to do the second step correctly. The third step is then completed correctly in one attempt. This is modelled as an experiment with two trials: Success-Fail-Success and Success-Success-Success respectively being the results of the two trials. Observe that we have created an "artificial second trial", where the successful steps from the first trial carry forward. Now consider the case where the thrid step also took two attempts. This is modeled as the following set of trials: Success-Fail-Fail and Success-Success-Success.

4.3 Building a Bayes Net Operator Model

Given a set of instructions, and commands that operators use to perform those insructions, we construct a Bayes Net as follows:

1. Construct a node for each instruction step (or comand directly accomplishing that step) and one for each command or action that assists in accomplishing that step.
2. Construct a directed edge between action A and action B if result of B depends directly on result of A.
3. Assign probability distributions corresponding to one attempt.

Figure 1 shows the instructions for the experiment we consider – adding an application (Tomcat) server to the existing infrastructure.

We construct a node each for copying, configuring, and starting Tomcat, and two nodes each for stopping, configuring and starting Apache (because there are two of them in our setup). We then add nodes corresponding to commands operators type to help them complete these steps (obtained from the bash history files of operators). We then add edges to show dependencies between instructions and between commands and instructions. Finally we add a node to indicate if the new application server has been integrated. We then specify conditional probability distributions for each node in the network. This is done based largely on experience in doing this experiment and also on bash histories of operators who performed this experiment. Figure 2 shows the resulting network.

This actually represents the first attempt at performing each of these steps. In subsequent attempts, operators may look at various log files, check process status, etc. We add these nodes to get the network in figure 3.

Depending on the number of trials, the slice shown in Figure 3 may repeat arbitrary number of times. We hence construct a Dynamic Bayesian Network extending the existing network as follows:

1. Construct an edge from each instruction step at one time instance to the same instruction step at the next time instance. We will specify conditional probabilities to encode the belief that if a step has correctly been performed once, it is performed correctly in the next trial(or slice).

2. Construct an edge between any two instruction steps in two time slices if performing one step in one slice helps perform the other step in the next slice (eg. configuring Apache and configuring Tomcat)

3. Specify conditional probabilities such that if the instruction was correctly performed in a previous slice it is correctly performed in the current slice, and if it was not correctly performed in the previous slice, conditional probabilities are the same as in a static Bayesian Network (ignore the edge from the previous slice).

4. Specify the number of trials.

This completes the specification of the Dynamic Bayesian Network. Figure 4 shows the first and second slice of such a dynamic network. For clarity, we have left out the *integrated* nodes at each time slice.

TASK: Add Application Server

Operator procedure:

To add an application server to skull15s.

1. Copy Tomcat from one of the nodes running Tomcat to the corresponding location on the new machine.
2. Modify jvmroute in conf/server.xml and conf/mendosus-server.xml.rubis.appstate to reflect the correct network interface.
3. Set the network interface in conf/mcast_heartbeat_db.conf to the same value.
4. Start tomcat using bin/startup.sh, you may have to use slide.
5. Verify if tomcat and the heartbeat processes are running.

---

6. For both webservers
   - Stop the webserver by running bin/apachectl stop (you may need slide to do this)
   - In conf/workers.properties, and conf/workers.properties.rubis add entries for the new application server.
   - Start the webserver by running bin/apachectl start.
   - Check to see if the webserver is up by looking for httpd processes.
7. Check apache logs to ensure requests are getting through to the new tomcat server.

End of operator procedure.

Figure 1: Instructions for Adding an Application Server

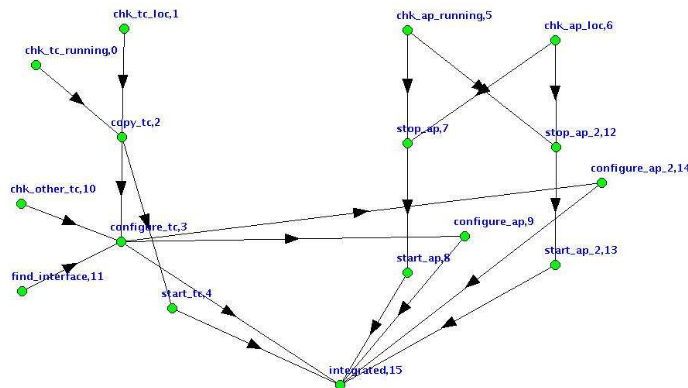

Figure 2: Building a Bayes Net for Adding an Application Server

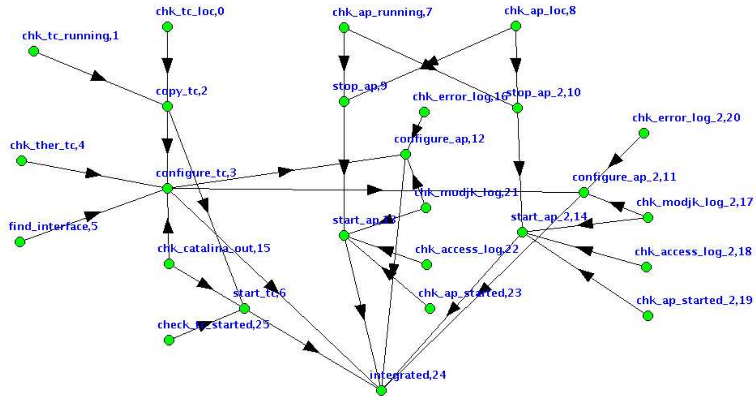

Figure 3: Extending the Bayes Net

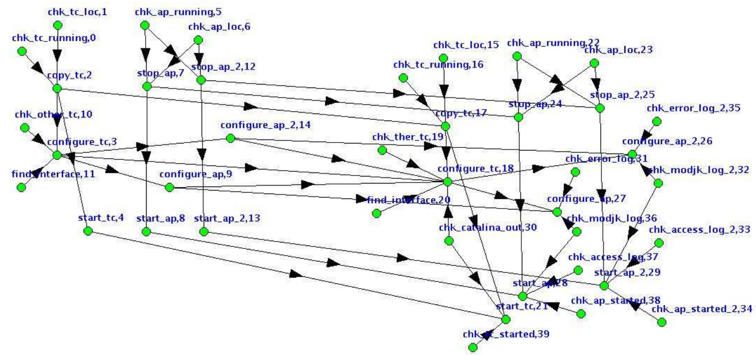

Figure 4: The Dynamic Bayes Net

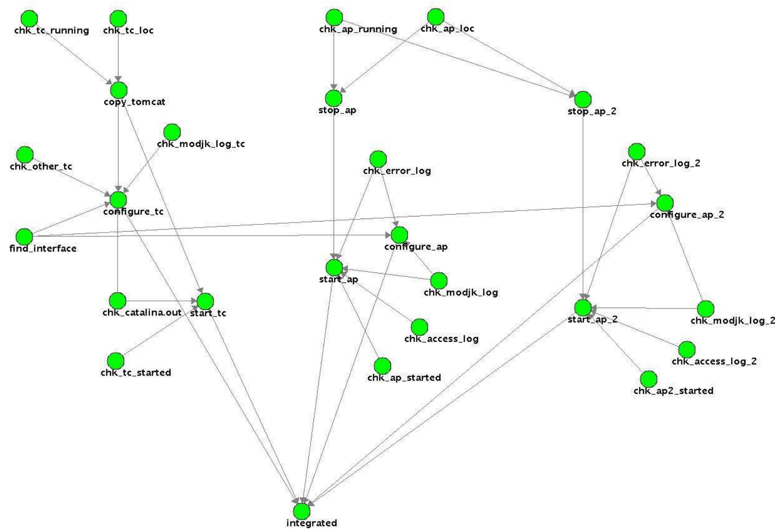

Figure 5: The Simplified Bayes Net

4.4 Simplifying the Bayes Net Operator Model

We may do inference with the Dynamic Bayesian Network just constructed, or reduce it to a static network. This is possible mainly because in the experiment model, once a step is performed correctly it is assumed to be performed correctly in all future trials (because the operator does not really do it again, we just assume it is done again). If it is not performed correctly, the conditional probability of doing it correctly in the next slice depends only on factors in the next slice (see section 4.3for algorithm to build the Dynamic Bayesian Network). We reduce the Dynamic Bayesian Network to a static one as follows:

1. Drop edges between an instruction step in one slice to the same step in the next slice.
2. For an edge between an instruction step in one slice and a different step in the next slice, include a common parent.
3. Specify conditional probabilities for each node (there are no parent nodes from previous slices)
4. Use any intermediate slice as a static Bayesian Network.

Figure 5 shows the result of performing the above steps on the dynamic network shown in Figure 4.

4.5 Doing Inference

We show results of doing inference with the static network in Figure 5 in Tables 2, 3, and 4. Since a Bayesian Network specifies the full Joint Probability Distribution of nodes in the network, we can ask any questions about these nodes. In Table 2, we see that the probability of correctly integrating the new server is 0.023 in a single trial, given that each step is done correctly with the probability specified in the network. Likewise, the probability of configuring Tomcat and Apache is 0.403 and 0.41 respectively. But given that the processes are running (those steps have been performed correctly), the probabilities go up slightly. If we also know that the interface has been correctly detected, see Figure 1, the probabilities are much higher. In Table 3 we see the importance of checking log files for configuring the servers. The probability of successful integration is more than doubled when the logs are checked compared to the case when they are not. In Table 4 we see the results of diagnostic inference, a reversal of the inference in table 3.

Finally, we can predict the number of trials operators would take to complete an experiment. We show this for the case where the processes have been successfully started and the interface has been correctly detected by the operator. From Table 2, we see that the probability of successful integration in one trial is 0.38. Therefore, for a 95% confidence in completing the experiment successfully, an upper bound on the number of future trials is

$$1 - (0.62)^{trials} \geq 0.95 \Rightarrow trials = 7$$

Note that this assumes the observed successful steps carry forward to future trials. Other steps are successful based on their posterior probability distribution computed from the network. Since we do not make any more observations, the success of those steps may not necessarily carry forward to future trials. Hence *trials* is an upper bound on the estimated number of future trials required to complete the experiment correctly.

4.6 Limitations of the Model

The number of experiments used to build the model are not statistically significant and the model has not been validated with other data. The operator model does not take into account help given by the person conducting the experiment to the operator. This help, while marginal for the experiment modeled, was not consistent across experiments. Operator background and skills were not factored in while deciding conditional probabilites - they were made based on judgement, and by averaging over all experiments.

5 Markon Decision Process Model

This section describes a model of the ongoing interaction between operator and system as a discrete-time Markov decision process. This model treats the operator as an agent that observes the state of the system and in turn selects actions on the system to change its state. System faults that occur spontaneously are also part of the state of the system that the operator must observe and repair. The model incorporates the reality that operators do not always perfectly observe the state of the system or act perfectly upon it.

The model also includes a methodology to use data gathered from operator simulation experiments to provide quantitative diagnostic conclusions about operator behaviors and their effects on the system as well as predictive capabilities to devise optimal operator actions in response to various system states and quantitative predictions about untried sequences of operator actions.

The determination of states and definition of operator actions have been gleaned from operator experiments on a particular system at Rutgers using Apache, Tomcat and MySQL. However, the methodology that determined those states and operator actions carries over into general three-tier Internet services and adapted to real-world situations.

Observe	Inference
NIL	Integrated:0.023
NIL	Config_tc:0.403
NIL	Config_ap:0.41
Apache and Tomcat Running	Integrated: 0.095, Config_tc: 0.47, Config_ap: 0.41
Apache and Tomcat Running and Interface detected	Integrated: 0.38, Config_tc: 0.9, Config_ap: 0.65
Integrated: Fail	Config_tc: 0.39, Config_ap: 0.4

Table 2: Inferring successful configuration and integration

Observe	Inference
Chk_modjk_log and Chk_access_log True	Integrated: 0.16, Config_tc: 0.44, Config_ap 0.7
Chk_modjk_log, Chk_access_log False	Integrated: 0.068, Config_tc: 0.39, Config_ap 0.33

Table 3: Inferring effect of logs on successful configuration and integration

5.1 Operator Actions and System States

The key issues for usefulness and applicability of the MDP model are the selection of system states and the determination of what constitutes the operator actions. The goal to both is to find the most useful granularity.

For system states, the number of actual client requests or the status of any client request are inappropriate factors in the determination of system state since this information would not lead to action on the part of the operator. Instead, factors that require the attention of and possible actions on the part of an operator are significant. Too coarse a granularity ("Is everything all right?") would provide no useful insight.

Similarly for operator actions, a focus on individual commands would not reliably capture operator intent (e.g. which editor an operator uses to modify a configuration file), when the desire is to make inferences about the goals of an operator's actions (e.g. modify a configuration file for a new server).

The state of the system is best defined in terms of the state of each machine in the system. The state of each machine is defined in terms of operator actions that must must performed on the machine. Also, operator actions were defined as changes to machine state brought about by precursor actions that did not directly affect the state of a machine by themselves. The precursor action of changing a configuration file would not by itself directly change the state of the system, but the act of bringing up a server that uses the configuration file would directly affect the state of the system. A precursor action like changing a configuration file might be done correctly or done incorrectly in a variety of ways. Examination of the commands and system data captured in the operator experiments as well as active observation of the operator subject during the experiments are required to determine what kinds of errors operators might make while performing a particular task. The selection of machine states comes from examination of the data gathered from the operator experiments.

We discussed different, hierarchical approaches to defining machine states, but none seemed to be clearly superior or viscerally consistent with intuition. There does not seem to be a fixed, general guideline as to how machine states should be selected. Different circumstances will lead to different state definitions depending on what the system evaluators regard as the most useful approach. How different operator errors observed in the data from operator experiments would be grouped together to define machine states must be left to the intuition of the system evaluators.

For the Add Application Server experiments, the states chosen represent some of the characteristics of our particular system set-up. The states chosen were specific to our system, but they do reasonably reflect general problems in three-tier systems. For annotation and data extraction purposes, the states were encoded, with a different encoding scheme for each tier of the system. The states used for the Add Application Server experiment appear below:

- wsD: web server machine intentionally down.
- ws1: web server machine operational but does not recognize all application servers.
- ws2: web server operational, but workers.properties misconfigured (last line unchanged).
- ws3: web server workers.properties misconfigured to cause mod_jk crash.
- ws4: webserver started from wrong place (usually /usr/bin).

Observe	Inference
Config_ap: False	Chk_modjk_log: 0.09, Config_tc: 0.38
Config_ap: True	Chk_modjk_log: 0.34, Config_tc 0.42

Table 4: Diagnosing cause of unsuccessful integration

- wsN: web server machine operating normally as expected.
- apD: application server intentionally down.
- ap2: application server started without proper slide/root permission.
- apN: application server operating normally as expected.

Notice that there was an ap1 state for the application tier, but it proved to not be useful and was removed after machines in the ap2 state had been discovered and used in the recorded operator data files.

The state of the system is encoded as the concatenation of how many machines are in each relevant state and the states of the machines in each tier. The machine states must appear in a consistent order, but the ordering requirement reflects a limitation of our state manipulation algorithms and is not inherent to the idea of system state encoding.

5.2 Getting MDP State Information

The goal is to translate information recorded during the operator experiments into state transition information for use by MDP algorithms. The timestamps in the .mbash_history files for the Add Application Server experiment were annotated by hand with operator actions, system state transitions and throughput levels in a compact format readable by an automated parser. An extraction tool was written to find and parse the annotations in the .mbash_history files and from them, construct the following:

- A state transition probability matrix;
- An action matrix associating actions with specific state transitions;
- A reward matrix associating throughput levels with specific state transactions;
- A list of all actions performed by the operators;
- A list of the mean duration of each system state defined in the annotations.

Action and reward information is taken directly from the annotations. The mean duration of each state is calculated from the timestamp associated with each state transition annotation. The state transition probability for each transition $s_i \Rightarrow s_j$ is calculated as the number of times that s_i transitions to s_j divided by the total number of transitions from s_i.

Operator interaction with the system is modeled as an MDP. The operator is the agent and the available actions are things that an operator might do while performing maintenance or fault correction tasks. The states are possible system states. The reward is the level of throughput associated with each operator action. An operator policy defines an action to take for every system state. There may be any number of policies of any origin.

Many of the calculations using the MDP model require the calculation of the equilibrium probability distribution using a matrix manipulation tool like Matlab.

5.3 Using MDP State Information

It is possible to use operator action and state transition information culled from operator experiment data to provide diagnostic and predictive evaluations to improve operator interaction with an existing three-tier system for maximum throughput. By modeling the operator interaction with a system as a Markov decision process, it is possible to make quantifiable calculations that can be used to minimize costs and maximize system throughput.

An initial concern was the proliferation of states when dealing with operator experiments covering a variety of tasks. There are several factors that tend to increase the number of states. Differentiating between similar operator errors leads to too fine a granularity of the state space, as does the necessity of counting machines in the system states. The number of states can be controlled by selecting sates with as coarse a granularity as individual circumstances will allow and focusing on single operator tasks whenever possible.

5.4 Comparison of Operator Policies

An operator policy may consist of a few simple steps having direct and obvious effects on total system throughput, or many steps whose individual and collective effects on system throughput may not be readily apparent. With an

MDP operator model, it is possible to determine a quantitative cost/reward of operator policies of arbitrary complexity. Once calculated, the cost/reward of different operator policies can be directly compared to determine the better policy.

The cost of an operator policy can be calculated with the following steps:

1. Select a policy to evaluate. Represent that policy as a state transition matrix.

2. Generate the equilibrium probability distribution for the state transition matrix.

3. Calculate the cost/reward of the policy as the sum of the products of equilibrium probability and throughput level for each state.

A policy evaluator tool was written that takes the equilibrium probability distribution and throughput level of each state as inputs and produces the cost/reward of the policy.

We treated the actions from a couple of annotated .mbash_history files for single, successfully completed operator experiments as if each represented a specific policy for the same Add Application Server task. Each operator completed the task with a different sequence of actions, and produced a quantified output for each policy.

5.5 Optimal Policy Determination

Many operator experiments over the same operator task will produce a variety of operator actions and system states, some of which may not have been anticipated prior to the actual experiments themselves. Each such action will have its own effects on system throughput. It has been shown that it is possible to mix and match individual operator actions into one or more policies that are optimal in the sense that they maximize throughput (the "reward" in MDP terms). Better still, such an optimal policy may be automatically generated using either policy iteration of value iteration algorithms [9].

There may be more than one optimal policy for a given situation. All optimal policies for a given situation will provide the same level of reward.

An optimal operator policy may be generated for each operator task by following these steps:

1. Use all available data from the relevant tasks.

2. Generate the cost/reward matrix for the tasks.

3. Generate the action matrix for the tasks.

4. Generate an optimal policy from the action and cost/reward matrices using either the policy iteration or value iteration algorithms.

An optimal policy generator tool was written that takes as input a list of actions, an action matrix and a cost/reward matrix for a task. The optimal policy generator uses the policy iteration algorithm to produce a list of optimal actions for every system state, in other words, an optimal policy.

A sample optimal policy was generated from the system state information from all of the annotated .mbash_history files for the Add Application Server experiment. The extractor tool was used to find the annotations and construct the list of operator actions, the cost/reward matrix and the action matrix which was piped directly into the optimal policy generator. The equilibrium probability distribution was calculated with Matlab.

In many of the operator experiments, the test operators managed to successfully complete the assigned task by repreatedly visiting only a subset of all the system states. For these states, the optimal policy generator produced intuitively sound results. When a single operator took the system into states not known before the experiment or later revisited in later experiments, but otherwise managed to complete the task successfully, the optimal policy generator produced results based on the single experience with those states. When operators took the system into system states unique to that experiment run and failed to complete the tasks successfully, the last timestamp of the corresponding .mbash_history file was annotated with an action that meant "gave up" and the extra time to correct completion of the task was added as part of the annotation. When given these unique system states with no alternative action that produced a higher reward, the optimal policy generator decided that the best action for these problematic states was, "give up and let somebody else fix it."

5.6 Determining The Proportion Of Time A System Spends In Each State

The evaluators of a system would be motivated to know the proportion of time the system spends in each state and the level of throughput associated with each state in order to quantify how much each suboptimal state costs in terms of lost transactions. The number of lost transactions would be the product of the reduction in the level of throughput associated with a system state times the length of time the system is in that state. Also of interest would be the operator actions that put the system into states that are particularly detrimental to overall throughput. Armed with the knowledge of what system states cost the most in terms of lost transactions, the system evaluators would be able to focus throughput improvement efforts on the states and actions that would lead to the greatest benefit.

The calculations require that experiment data exist for all operator maintenance and fault repair tasks. Maintenance tasks occur on a regular schedule or at least at some known

historical frequency for the system under observation. Similarly, fault correction tasks occur with some historically observed frequency. The time spent in the “normal” state for each task would have to be adjusted according to the frequency of each task.

Operator experiments can end with the test system left in a problematic state, but in the real world, any remaining problem must be fixed. The time spent in the problematic state at the end of an operator experiment must be extended to the time when the problem would actually be corrected. At this time, how much time to add is more art than science and no clear objective principle exists.

The relative proportion of time a system spends in each state can be calculated by following these steps:

1. Use experiment data from the operator experiments for all tasks.

2. Generate the equilibrium probability distribution for the state transition matrix.

3. Determine the mean time spent in each state from the experiment data.

4. Determine the level of throughput for each state from the experiment data.

5. Generate the relative proportion of time spent in each state from the equilibrium probability distribution and the mean time spent in each state.

An relative proportion generator tool was written that takes as input a list of states, the equilibrium probability distribution and the mean time spent in each state from the operator experiment data. The output is the relative proportion of tie the system spends in each state.

This method was tested by using the system state information from all of the annotated .mbash_history files for the Add Application Server experiment. The time spent in the “normal” state was increased to represent the addition of an application server exactly one a week. Matlab was used to generate the equilibrium probability distribution.

6 Future Work

Our proof-of-concept operator modeling and experiment infrastructure could be extended in three aspects: Creating and runinning statistically valid experiments, adjust probabilities and time intervals in the Markov decision model, and improve tools for generating the Markov matrices.

The ultimate, and probably ambitious, culmination of this piece of work would be the development of tools to assist operators based on the captured operator model.

6.1 Experiments

The fact that we were not able to collect statiscally significant data is, undoubtedly, a major flaw in our study. We did not even try to analyze the data that we gathered by means of some statistical techniques such as the so-called *paired-t analysis* which a study aimed at a maintainability benchmark for RAID systems makes use of [5]. These are hence two important steps to be taken next.

In addition, we shall extend our experiments to more complex systems. Even though we could claim that our three-tier architecture is representative of a typical Internet service, the very fact that we have a testbed with only eight server machines tremendously limits the complexity of the operator tasks.

Although we have interesting results for five types of experiments, we modeled only one. Clearly, we shall apply the Bayesian network and the MDP approaches to model the remaining experiments.

6.2 Model

For Markov Decision Processes, system evaluators may want to play ”What if ...?” games with the probabilities generated from the experiment data in order to determine the gains or losses incurred if specific actions are or are not taken when the system is in a particular state. Probabilities could be shifted, entire actions eliminated or previously unchosen actions selected. The only constraint is that the probabilities in any row of the adjusted matrix must add up to 1 to preserve the basic property of a state transition probability matrix. Once the state transition probability matrix has been adjusted, the system evaluators would be able to determine the relative proportion of time the system would spend in each state if the modified actions represented in the state transition probability matrix were realized. Similar adjustments could serve as an alternate approach to comparative policy evaluation, where new policies are created on an ad hoc basis.

6.3 Tools

The extractor tool for the Markov Decision Process could be modified to calculate the equilibrium probability distribution directly from the state transition probability matrix without the extra step of using external matrix manipulation tools such as MatLab.

The .mbash_history files were annotated by hand. The annotation was very time consuming. The annotator spent much of the time in determining the state of the system from what the operator had done to the system prior to either bringing up or taking down a server. The state transition algorithms in the extractor implementation know nothing

about individual annotations and the exact determination of system state is accordingly difficult. Additional annotation of what the operator did that did not directly affect the state of the system could be added that would allow an enhanced extractor tool to determine the state of the system from the simple annotations indicating what the operator had done prior to acting directly on a server.

Also, the optimal policy generator implementation does not understand the idea of reversing an action, which lead to the optimal policy genertor being unable to provide a useful course of action for problematic system states where no operator provided a course to successful completion of the task. If the optimal policy generator could instead reverse operation sequences that lead to such problematic states, then an optimal policy generator would be able to proivde steps that would return the system to states from which there is a course to successful completion of the task.

7 Conclusion

Operator modeling is an important step for systems community to take in order to create complex systems that function smoothly. With this study, we contribute a proof-of-concept infrastructure for running experiments with human test subjects and show that it is possible to transform the results into different operator models. We also provide sample procedures in running those experiments and gathering data from them.

Time and resource constraints prevent us from gathering enough data to make statistically significant conclusions with regard to the frequency and effects of operator actions and errors. It is also not clear how such a model could apply to distributed systems in general. Even so, we produced both a Bayesian network and a Markov Decision matrix specific to a three-tier Internet service that demonstrates the usefulness of our infrastructure. The network gives snapshots of how operator actions affect our system, and the matrix offers optimal decision policies with which to traverse the system. Both these operator models can be extended to present system designers and operators with knowledge of how a particular system would react with human operators under a given circumstance.

8 Acknowledgments

We are truly thankful to everybody who volunteered to participate in our experiments, namely Wenyuan Xu, Chris Gates, Prashant Mekaraj, Andrew Tjang, John McCabe, Nitin Gupta, Martin Constantine, Minyoung Kim, Arati Baliga, Abhishek Mehrotra, Andrey Kravtsov, Lev Kaufman, Vishal Shah and Nitin Shetti. They made this work possible.

We are also indebted to Professors Thu Nguyen, Michael Littman, Ricardo Bianchini and Vladimir Pavlovic for the fruitful discussions that helped us improve the quality of this work.

Last but not least, we would like to express our gratitude to Kiran Nagaraja who provided us with both Mendosus and a preliminary version of the monitoring infrastructure we used to log the operator actions.

References

[1] A. Brown, L. Chung, W. Kakes, C. Ling, and D. A. Patterson. Dependability benchmarking of human-assisted recovery processes. In *To appear in the Proceedings of the IEEE International Conference on Dependable Systems and Networks (DSN 2004)*, June 2004.

[2] A. Brown, L. Chung, and D. A. Patterson. Including the human factor in dependability benchmarks. In *Proceedings of the DSN Workshop on Dependability Benchmarking*, June 2002.

[3] A. Brown and D. Patterson. To err is human. In *Proceedings of the First Workshop on Evaluating and Architecting System dependabilitY (EASY '01)*, July 2001.

[4] A. 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.

[5] Aaron Brown. Towards availability and maintainability benchmarks: a case study of software RAID systems. Master's thesis, Computer Science Division-University of California, Berkeley, 2001.

[6] Dynaserver: System support for dynamic content web servers. Available at http://www.cs.rice.edu/CS/Systems/DynaServer/, February 2004.

[7] Xiaoyan Li, Richard P. Martin, Kiran Nagaraja, Thu D. Nguyen, and Bin Zhang. Mendosus: A SAN-Based Fault-Injection Test-Bed for the Construction of Highly Available Network Services. In *Proceedings of the 1st Workshop on Novel Uses of System Area Networks (SAN-1)*, Cambridge, MA, January 2002.

[8] Michael L. Littman, Nishkam Ravi, Eitan Fenson, and Rich Howard. An Instance-based State Representation for Network Repair. In *To appear in the Proceedings*

of the 19th National Conference on Artificial Intelligence (AAAI), 2004.

[9] Tom Mitchell. *Machine Learning*. McGraw-Hill, 1997.

[10] David Oppenheimer, Archana Ganapathi, and David Patterson. Why do Internet services fail, and what can be done about it. In *Proceedings of the Usenix Symposium on Internet Technologies and Systems*, March 2003.

[11] 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, March 2002.