Probabilistic Model-Driven Recovery in Distributed Systems

Kaustubh R. Joshi, *Member, IEEE,* Matti A. Hiltunen, *Member, IEEE,* William H. Sanders, *Fellow, IEEE* and Richard D. Schlichting, *Fellow, IEEE*

***Abstract*—Automatic system monitoring and recovery has the potential to provide effective, low-cost ways to improve dependability in distributed software systems. However, automating recovery is challenging in practice because accurate fault diagnosis is difficult given the common monitoring tools and techniques with low fault coverage, poor fault localization, detection delays, and false positives. In this paper, we present a holistic model-based approach that overcomes these challenges and enables automatic recovery in distributed systems. To do so, it uses theoretically sound techniques including Bayesian estimation and Markov decision theory to provide controllers that choose good, if not optimal, recovery actions according to a user-defined optimization criteria. By combining monitoring and recovery, the approach realizes benefits that could not have been obtained by using them in isolation. We experimentally validate our framework by fault injection on realistic e-commerce systems.**

***Index Terms*—Fault tolerance, Monitoring, Diagnosis, Distributed systems, Application-aware adaptation, POMDP**

I. Introduction

Building highly available systems has always been critical in certain areas, but society's increased reliance on distributed systems for applications such as search, entertainment, and e-commerce requires low-cost solutions for high availability. Approaches for high availability in such settings are typically based on the combination of redundancy and 24/7 operations support in which human operators detect and repair failures and restore redundancy before the service is compromised. However, putting human operators in the critical path for availability is not ideal. To address this, techniques to recover failed hardware and software components automatically through restart and other methods have been proposed, including software rejuvenation [1], recursive restartability [2], and recovery-oriented computing [3].

While promising, automation of system recovery in realistic distributed systems can be complicated, and addressing these complications is the core contribution of this paper. One significant challenge is that, before recovery actions can be taken, the system must determine which components have failed, something that can be very difficult in real distributed systems in which monitoring is often done using multiple separate monitoring systems and techniques. For example, while some off-the-shelf components may provide interfaces for standards-based monitoring (e.g., SNMP [4]), other components may have no monitoring or provide only some proprietary methods. Although such component monitoring may provide good localization, it often does not provide an accurate picture of availability and system health as perceived by its users. Therefore, it is often augmented by end-to-end monitoring to verify that the system is able to provide its service as specified. However, end-to-end mechanisms are often geared to alerting the operators who then have to use their domain knowledge and other tools to determine the problem and the appropriate corrective actions. Monitoring mechanisms may also provide conflicting information and suffer from the typical problems of false positives (e.g., due to timeout-based testing) and low coverage. Furthermore, even when the fault can be narrowed down to a small set of candidate components, a system may allow a number of possible recovery or reconfiguration actions, and it is not easy to determine which sequence of actions would achieve the quickest recovery.

In this paper, we develop a holistic approach to automatic recovery in distributed systems using a theoretically well-founded model-based mechanism for automated failure detection, diagnosis, and recovery that realizes benefits that could not be achieved by performing them in isolation. In particular, combining the recovery actions with diagnosis allows the system to diagnose and recover from fault scenarios that would not be identifiable using diagnosis only, and to invoke additional monitoring only when it would help in choosing the right recovery action. Our approach works with imperfect monitoring systems that may already be present in the target system and with any application-specific recovery actions available to it. The approach combines Bayesian estimation and Markov decision processes to provide a recovery controller that can choose recovery actions based on several optimization criteria. Our approach has the ability to detect when a problem is beyond its diagnosis and recovery capabilities, and thus to determine when a human operator needs to be alerted. We believe that the approach is applicable to a wide variety of practical systems, and experimentally illustrate its use with two examples: (1) a small but realistic deployment of an Enterprise Messaging Network (EMN) platform developed at AT&T [5] and (2) a deployment of RUBiS [6], an open-source eBay-like auction system.

II. Overview

First, we motivate the issues involved in constructing an automatic recovery system using the AT&T Enterprise Messaging Network (EMN) as an example and describe a specific EMN deployment scenario that is used as a running example throughout the rest of the paper. We then provide a brief overview of our proposed approach.

Motivating Example. EMN is a platform for providing services to a wide range of mobile devices such as cell phones,

K.R. Joshi, M.A. Hiltunen and R.D. Schlichting are with AT&T Labs Research, Florham Park, NJ 07932. Email: {kaustubh,hiltunen,rick}@research.att.com

W.H. Sanders is with the University of Illinois at Urbana-Champaign, Urbana, IL 61801. Email: whs@illinois.edu

pagers, and PDAs via multiple protocols such as WAP, e-mail, voice, or SMS. Its architecture is representative of many modern multitier enterprise systems, and is shown in Figure 1(a). The architecture consists of several replicated software components, some of which are proprietary (e.g., front-end protocol gateways and back-end application/EMN servers), and some of which are COTS (e.g., a database, load-balancers, protocol servers, firewalls, and JMS (Java Message Service) servers). High service availability is a goal for the EMN platform, and rapid automated fault recovery would provide both cost and availability benefits.

The primary source of problem detection used by operations staff in an EMN deployment is an automated monitoring system that is based on a combination of monitors. End-to-end service monitoring verifies that the system as a whole is operational by submitting test requests to the system via the front-end protocol servers and verifying that the system generates valid replies. However, because each test involves a number of individual components, it is often impossible for the monitor to pinpoint exactly which individual component was faulty if the test fails. Furthermore, due to internal redundancy, the same test may sometimes succeed and sometimes fail simply because the test request may take different paths through the system. On the other hand, component monitoring is achieved via "ping" queries initiated by the monitoring system, or via periodic "I'm alive" messages sent by the software components. Although such monitors are able to pinpoint a faulty component precisely, they cannot test functional behavior and miss non-fail-silent failures in the components.

Once a failure has been detected and diagnosed, operators choose from a number of available recovery actions that include restarting of software components or hosts, creating additional copies of certain components for fault masking, reconfiguration of software, or physical repair of system hardware. Automatic choice of the right actions based on the fuzzy and sometimes conflicting picture of the system painted by the monitors is a challenging task that can have a large impact on availability.

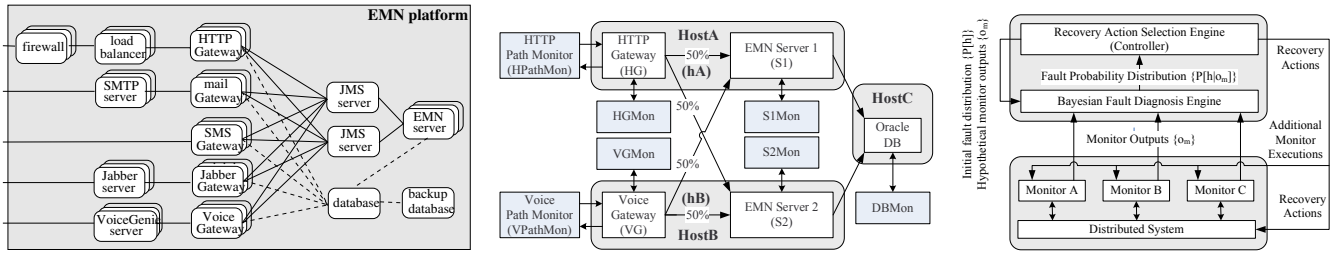
To highlight the key challenges and solutions for automatic recovery without an overwhelming level of detail, we illustrate our approach on a simplified though realistic example configuration of EMN shown in Figure 1(b). The configuration implements a Company Object Lookup (CoOL) system. The CoOL system allows look-up and update of current inventory levels and sales records, either via the company's website, or via phone. In addition to protocol gateways, the system consists of duplicated back-end EMN servers and a database housed on three hosts as shown in the figure. The gateways use round-robin routing to forward incoming requests to one of the two EMN servers, which then look-up and update the relevant data from the database and send a response back through the same gateway. The monitoring system contains component monitors for each component, and two end-to-end service monitors that test the overall functionality of the system via the two protocol gateways. Finally, the only two classes of recovery actions we consider in this paper are restarting of components and the more expensive rebooting of hosts.

The CoOL system is architecturally similar to many multitier systems that consist of web, application, and database servers. Additionally, the monitoring techniques used are two of the most commonly used monitoring techniques in enterprise applications such as those managed by AT&T for Fortune 500 customers. Designing monitoring techniques to provide good coverage without being intrusive is an important problem, but is orthogonal to our work. Our approach can use monitors that are already present in many systems irrespective of how they were designed. Finally, even though restarts and reboots alone cannot recover from all possible problems in a system, they are common actions employed by system operators, and are effective against temporary failures that are widespread in modern systems. Therefore, even though the recovery algorithms developed in the paper are completely general, their application to the CoOL example extends directly to a large class of practical e-commerce and business processing systems (e.g. web-services).

Approach Overview. The overall goal of our approach is to diagnose system problems using the output of any existing monitors and choose the recovery actions that are most likely to restore the system to a proper state at minimum cost. This is done via a *recovery controller* whose runtime architecture is presented in Figure 1(c). To combat individual monitor limitations, the recovery controller combines monitor outputs in order to obtain a unified and more precise probabilistic picture of system state. Any remaining uncertainty in system state is resolved by performing recovery actions and observing their effects. Overall, the approach consists of the following steps.

First, we determine which combinations of component faults can occur in the system in a short window of time. Each such fault combination is characterized by a fault hypothesis. (e.g., "Server 1 has crashed"). Then, we specify the coverage of each monitor m in the system with regard to each fault hypothesis h, i.e., the probability that m will report a failure if h is true (Section III). Finally, we specify the effects of each recovery action according to how it modifies the system state and fault hypothesis (e.g., restart of a component may convert a system where the "temporary fault" hypothesis is true into a system where the "no fault" condition is true).

During runtime, when a failure is reported by one or more of the monitors, the recovery controller is invoked. The controller operates in a series of steps. Each step is a complete monitor-decide-act cycle where the monitors are executed, their outputs are combined, and a recovery action is chosen and executed. The controller uses Bayesian estimation along with coverage models to combine monitor outputs and determine the likelihood for each fault hypothesis (Section IV). It then invokes a recovery algorithm to choose appropriate recovery action(s). We present two algorithms to choose appropriate actions: SSLRecover and MSLRecover (Section V). SSLRecover (Single Step Lookahead) is a simple greedy procedure that chooses actions that recover from the most likely fault. MSLRecover (Multi-Step Lookahead) is a more sophisticated algorithm based on partially observable Markov decision processes (POMDPs) that looks multiple steps into the future and allows optimization over entire sequences of recovery actions. It can also decide when additional monitoring is needed.



(a) EMN Architecture (b) Simple EMN Configuration (c) Automatic Recovery Architecture

Fig. 1. Approach motivation and overview

III. System Model

A. Fault Hypotheses

The recovery controller maintains (probabilistic) information about the presence of undiagnosed faults in the system through a set of fault hypotheses $\mathcal{H}$. A *fault hypothesis* $h \in \mathcal{H}$ is a Boolean expression that, when true, indicates the presence of one or more faults in the system. In the CoOL example, we define a "Down" fault hypothesis for hosts and two fault hypotheses, "Crash" and "Value" for components. Down(r) implies that host r has crashed[1] and does not respond to monitoring tests, but rebooting the host will fix the problem. Crash(c) means that component c has crashed, while Value(c) means that component c is alive, but does not provide correct service (e.g., it has hung and produces no result or the result is incorrect). Fault hypotheses are considered as opaque identifiers, but their names can reflect useful attributes such as the set of components associated with the fault hypothesis, the fault model, and the type of fault: temporary or permanent. The recovery controller also maintains a special null hypothesis, h_ϕ, that indicates that no faults are present in the system.

We assume that only one fault hypothesis can be true at a time, but this is not a limitation in practice since each fault hypothesis can represent multiple faults. For example, in the CoOL system, we could define a hypothesis Crash(HG, DB) indicating that both the HTTP Gateway and the database have crashed. We call such a fault hypothesis that denotes multiple component failures a *composite hypothesis*. Often, hidden inter-component dependencies and fault propagation cause cascading failures in distributed systems. We assume that the controller deals with such situations *after* the cascading has occurred. This allows us to model such failures just like other multi-component failures (e.g., common mode failures) using composite fault hypotheses. Conversely, a single component may have multiple fault hypotheses, each corresponding to a different failure mode. A single fault hypothesis may also cause multiple error messages or alarms, thus giving the appearance of cascading. However, such situations are dealt with using notions of monitors and coverage as described next.

B. Monitors

In practice, most systems detect faults either through external monitoring systems or error messages produced by the system's components themselves. We uniformly model both situations using the notion of monitors $\mathcal{M}$ that provide the only way for the recovery controller to test the validity of the various fault hypotheses. Each monitor returns true if it suspects a fault, and false otherwise. A monitor's output naturally depends on faults activated in the system, i.e., the set of fault hypotheses the monitor is able to detect, but we assume that it is independent of the outputs of other monitors and its previous outputs of the same monitor if invoked multiple times. We denote the set of fault hypotheses a monitor m monitors by $\mathcal{H}_m \subseteq \mathcal{H}$.

A system may include a variety of monitoring techniques including some of the following:

Heartbeat-based monitors in which lack of a timely heartbeat message from the target indicates failure.

Test-based monitors which send a test message to the target and wait for a reply. Messages may range from OS-level *pings* and SNMP queries to application level testing, e.g., a test query to a database object.

End-to-end monitors which emulate actual user requests. Such monitors can identify that a problem is somewhere along the request path but not its precise location.

Error logs of different types that are often produced by software components. Some error messages can be modeled as monitors which alert when the error message is produced.

Statistical monitors which track auxiliary system attributes such as load, throughput, and resource utilization at various measurement points, and alarm if the values fall outside historical norms.

Diagnostic tools: such as filesystem and memory checkers (e.g., fsck) that are expensive to run continuously, but can be invoked on demand.

The CoOL system has five ping-based monitors: HGMon, VGMon, S1Mon, S2Mon, and DBMon for the HTTP gateway, Voice gateway, EMN Servers 1 and 2, and the database, respectively. It also uses two end-to-end monitors HPathMon and VPathMon that test client visible functionality through the HTTP and Voice gateways, respectively.

C. Monitor coverage

The controller does not need to know how the monitors work, but it needs a specification of their ability to detect the various fault hypotheses. The specification is provided in terms of *monitor coverage*, P$[m|h]$, that represents the probability that monitor m will return true given that fault hypothesis $h \in \mathcal{H}_m$ is true. While such coverages have classically been deduced through fault injection (e.g., [7]) or failure log analysis (e.g., [8]), it is also possible to specify them

[1]The term "fault hypothesis" is used for component failures because they are faults from the whole-system point of view.

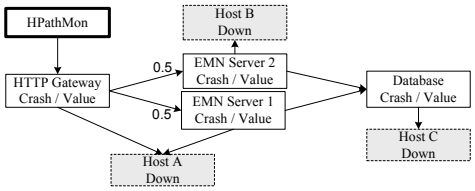


Fig. 2. Example monitor coverage graph

constructively. The monitor coverage of a simple monitor that targets one (or few) components is easy to understand and specify. For example, a monitor that pings a host in a local area network would detect failures of the host being pinged as well as the network link between the monitor and the target host. However, many monitors, especially the the end-to-end monitors, may test a large number of software components and processes, network links, and the hosts that are running the software.

To make it possible to describe complex monitor coverage, we use a *monitor coverage graph* that can be used to compute the coverage probabilities P$[m|h]$. Figure 2 illustrates the monitor coverage graph for the HTTP path monitor (HPathMon) in the CoOL system. Here, the nodes represent the fault hypotheses tested by the monitor. Since this monitor will detect both Crash and Value faults in the components, we present both fault hypotheses for each component as one node. An edge from node a to node b with weight w indicates that if a is tested by the monitor then b is also tested with probability w. The graph can also contain And-Or edges and indicate if a monitor relies on the outputs of other monitors. A formal description of the monitor coverage graph formalism can be found in [9]. The monitor coverage graph can be constructed by following the flow of the test request through the system and noting that testing a software component on a host also means that the host will be tested. For example, an HPathMon request always detects Crash and Value faults in the HTTP Gateway and the Database regardless of the path taken, but will only detect these faults in EMN Servers 1 and 2 some fraction of the time (0.5 when using round-robin load balancing).

Since monitors are often based on deviations from "expected" behavior, they may produce false positives. These can be specified simply as the probability that monitor m reports true when either the null fault hypothesis h_ϕ is true, or a fault hypothesis not in m's set of detectable hypotheses $\mathcal{H}_m$ is true, i.e., P$[m|h], h = \mathrm{h}_\phi \vee h \in \mathcal{H}_m$. The probability of false positives depends on a number of factors, including the timeout values used, load of the system, and the response time (or heartbeat) distribution. Since all of these factors may not be known (or cannot be empirically measured), providing an accurate probability for false positives can be difficult. Fortunately, a rough estimate of the coverage is usually sufficient. In Section VI-C we examine the sensitivity of the diagnosis to inaccuracies in the coverage values.

Table I shows the monitor coverage for a subset of the fault hypotheses in our CoOL system example. Because the coverage of a monitor for a fault hypothesis that it cannot detect is 0 and most monitors detect only a small subset of the faults, the table is sparsely populated. In this table, we ignore the false positives caused by timeouts that are too short. Notice that Value faults are not fail-silent, and therefore cannot be detected by ping-based monitors. However, end-to-end monitors can detect such application specific behaviors. Finally, we observe that, because of load balancing, none of the monitors can differentiate between the Value faults in the two EMN servers, thus making it impossible to determine the true faulty component just by diagnosis alone.

TABLE I
PARTIAL MONITOR COVERAGE

Monitor	Down	Crash			Value		
		HG	S1	S2	HG	S1/ S2	DB
HPathMon	1	1	0.5	0.5	1	0.5	1
VPathMon	0.5	0	0.5	0.5	0	0.5	1
HGMon	1	1	0	0	0	0	0
VGMon	0	0	0	0	0	0	0
S1Mon	1	0	1	0	0	0	0
S2Mon	0	0	0	1	0	0	0
DBMon	0	0	0	0	0	0	0

D. Recovery Actions

The application-specific recovery actions $\mathcal{A}$ provide the only way for the controller to change the truth value of fault hypotheses. An action a is specified in terms of its "fault hypothesis effect" function $a.\mathrm{hEffect}(h) \rightarrow \mathcal{H}$, its mean duration $a.\bar{t}(h)$, and the set of monitors $a.\mathcal{M}$ to be invoked after its execution. Permitting actions to choose monitors allows for "information gathering" actions that don't perform recovery, but improve diagnosis (e.g., diagnostic tools such as *fsck*). Such actions are automatically chosen by the controller when the usefulness of the output their monitors provide is balanced against their cost of execution. The hEffect function specifies how the execution of action a will transform fault hypothesis h if it is true. E.g., DB.Restart.hEffect(Value(DB))$\rightarrow$ h_ϕ indicates that if the DB is restarted in a situation in which it has a Value fault, the fault will be repaired while DB.Restart.hEffect(Crash(HG,DB))$\rightarrow$Crash(HG) indicates that restarting the DB when both the HTTP server and DB have crashed will still leave the HTTP server in crashed state. Recovery actions may cause additional faults to occur. E.g, restarting a server might induce disconnections in other servers that communicate with it as specified by Apache.Restart.hEffect(Fault(Apache)) $\rightarrow$ TempFault(Tomcat). However, we assume that there is some sequence of actions that will eventually recover from all the fault hypotheses included in the models. We restrict the recovery actions to have a deterministic effect because re-execution of a recovery action typically does not have a different impact unless something else in the system has changed. However, probabilistic action outcomes can be easily added if needed [9] without major changes to any algorithms.

The CoOL example uses only two types of recovery actions. c.restart() actions allow recovery from Crash(c) and Value(c) faults by restarting of component c, while r.reboot() actions allow recovery from Down(r) faults by rebooting of host r. We assume that all the faults are temporary ones that are fixed by restarting. However, other types of prepackaged recovery actions can be included. E.g., rollback actions that restore files or process state from backups can recover from permanent state damage (e.g., corrupted configuration files, damaged tables) and can be modeled as

Rollback(ServerConfigFile).hEffect(ServerConfigFault) $\rightarrow$ h$_\phi$. Other permanent failures (e.g., hardware failures) can be recovered from by failing over/migrating to a designated backup host. Additional examples can be found in [9]. The advantage of using the recovery framework for such prepackaged actions is that it allows very different types of monitors at different levels of the software stack ranging from the OS to the application to drive the actions. Even if two different types of faults have indistinguishable effects (as is the case in the CoOL example for EMN Value faults), the framework will intelligently choose actions based on their costs and eliminate fault hypotheses based on the outcomes of these actions.

IV. Probabilistic Diagnosis and Recovery

Next, we describe the Bayesian diagnosis step used to combine monitor outputs and show how the recovery algorithm works without going into the details of action selection.

A. Bayesian Diagnosis

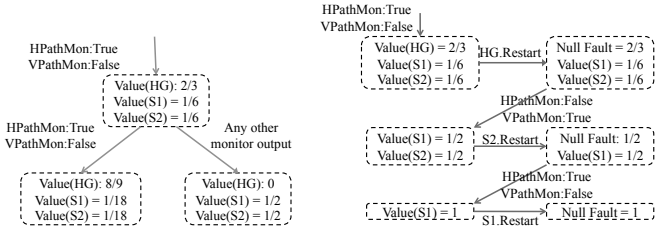
The idea of using Bayesian estimation to deal with incomplete monitor information is not new. However, past work on fault diagnosis (e.g., [10]) has focused mainly on tests of a single type. We believe that our uniform treatment of multiple types of monitors with differing characteristics, together with the additional diagnosis capabilities arising from the coupling of diagnosis and recovery differentiates our work from previous diagnosis efforts.

In the following discussion, let $\mathcal{M}' \subseteq \mathcal{M}$ be the subset of monitors invoked in the current round, o_m denote the current output of monitor m, and $o_{\mathcal{M}'}$ be the current set of all monitor outputs. Let P$[h]$ be the apriori probability that fault hypothesis h is true. The vector $\{\mathrm{P}[h]|h \in \mathcal{H}\}$ is the *diagnosis vector*. If the controller has no prior knowledge about the frequency of faults, it can assume they are equally likely and set $\mathrm{P}[h] \leftarrow 1/|\mathcal{H}|, \forall h \in \mathcal{H}$. Using the monitor outputs and coverage models $P[m|h]$, a new diagnosis vector $\mathrm{P}[fh|o_{\mathcal{M}'}]$ can be calculated using the Bayes rule as follows.

$$\mathrm{P}[h|o_{\mathcal{M}'}] = \textbf{Bayes}(\mathrm{P}[h], o_{\mathcal{M}'}) = \frac{\mathrm{P}[o_{\mathcal{M}'}|h]\mathrm{P}[h]}{\sum_{h' \in \mathcal{H}} \mathrm{P}[o_{\mathcal{M}'}|h']\mathrm{P}[h']} \quad (1)$$

$$\mathrm{P}[o_{\mathcal{M}'}|h] = 1.0 \times \prod_{\mathcal{M}'_h} \mathbf{1}[o_m]\mathrm{P}[m|h] + \mathbf{1}[\neg o_m](1 - \mathrm{P}[m|h]) \quad (2)$$

Here, $\mathbf{1}$[expr] is the indicator function, and is 1 if expr is true, and 0 otherwise. The product term in Equation 2 is due to the monitor independence assumption. Only monitors that are invoked in the current round, and which cover h are considered in the product, i.e., $\mathcal{M}'_h = \{m|m \in \mathcal{M}' \wedge h \in \mathcal{H}_m\}$. If no invoked monitors cover a fault hypothesis h, $P[o_{\mathcal{M}'}|h]$ defaults to 1 (i.e., no change in likelihood). Note also that if the original hypothesis probability P$[h]$ is 0, then so is the updated hypothesis probability. We call this the *zero preservation* property. Hence, if a fault hypothesis is known to be false (e.g., a recovery action that recovers from that fault was executed earlier), then the initial hypothesis probability can be set to 0 without fear of it becoming non-zero after one or more Bayesian updates.



(a) Diagnosis (b) Recovery

Fig. 3. Diagnosis and Recovery in the CoOL Example

Input: $o_{\mathcal{M}'}, \{\mathrm{P}[h]\}, \mathcal{H}, \mathcal{M}, \mathcal{A}, \epsilon$
while *true* **do**
  $\{\mathrm{P}[h]_{old}\} \leftarrow \{\mathrm{P}[h]\}$; $\{\mathrm{P}[h]\} \leftarrow \{\textsc{Bayes}(\mathrm{P}[h], o'_{\mathcal{M}})\}$
  if $\nexists h \in \mathcal{H}$, *s.t.* $P[h] > \epsilon$ **then return** *alert*
  if $P[h_\phi] \geq 1 - \epsilon$ **then return** *success*
  $a \leftarrow \textsc{ChooseAction}(\{\mathrm{P}[h]\}, \mathcal{H}, \mathcal{M}, \mathcal{A}, \epsilon)$
  $\textsc{ExecAction}(a)$
  foreach $h \in \mathcal{H}$ **do**
    $p' \leftarrow \mathrm{P}[h]$; $\mathrm{P}[h] \leftarrow 0$;
    $\mathrm{P}[a.\mathrm{hEffect}(h)] \leftarrow \mathrm{P}[a.\mathrm{hEffect}(h)] + p'$
  $\mathcal{M}' \leftarrow a.\mathcal{M}$; $o_{\mathcal{M}'} \leftarrow \textsc{ExecMon}(\mathcal{M}')$

Algorithm 1: Recovery Algorithm

CoOL Example. Figure 3(a) shows the monitor outputs and resulting fault hypothesis probabilities when HPathMon reports true (i.e., abnormal), but all other monitors report false in the CoOL system. Plugging the monitor outputs into Equation 1 eliminates all Down and Crash fault hypotheses (the Ping monitors were normal) along with Value(DB) and Value(VG) (since VPathMon worked). The data is not enough to distinguish between Value(HG), Value(S1), and Value(S2), but hints towards Value(HG) (since the VPathMon test, which also went through the EMN servers did work). An additional round of monitoring sheds more light on the situation. If the same monitor results are observed again, the evidence strongly favors a Value(HG) fault. However, if HPathMon now reports a normal result, that eliminates Value(HG), and throws suspicion on Value(S1) or Value(S2). The same thing happens if both path monitors return true (since only one fault hypothesis can be true at a time, a failed VPathMon test implies that it cannot be Value(HG)). Since the two EMN servers are indistinguishable from the point of view of path monitors, the diagnosis cannot be further improved.

B. Recovery Algorithm

Recovery is an iterative processes of repeated diagnosis and action that is shown in Algorithm 1. The process begins when the recovery controller is invoked with an initial set of monitor outputs $o_{\mathcal{M}'}$ that indicate a failure, and an initial diagnosis vector $\{\mathrm{P}[h]\}$. Using these inputs, the controller performs a Bayes update on the initial diagnosis vector, and subsequently uses two conditions to decide if recovery should be terminated. First, recovery is terminated with an operator alert if the probabilities of all the fault hypotheses approach zero. This occurs in the Bayesian update when the monitor observations cannot be explained by any fault hypothesis that has a non-zero probability, and usually indicates either a missing fault hypothesis, or incorrect monitor or action models. In this situation, the controller stops rather than continuing operation with bad

models. Second, if the probability of the null fault hypothesis approaches 1 (within ϵ), the algorithm considers recovery to be a success and terminates. Successful termination means that the fault was recovered with an arbitrarily high confidence (tunable by choice of ϵ).

The algorithm then uses the diagnosis vector to choose and execute an appropriate recovery action a using the techniques described in Section V. Subsequently, it updates the diagnosis vector to reflect the effects of the action by transferring the probability of each fault hypothesis h to the new fault hypothesis specified by the action's $a.\text{hEffect}(h)$ function (e.g., h_ϕ if the action recovers from the fault h). The recovery action informs the controller of the monitors $a.\mathcal{M}'$ which are to be invoked in the next round. The controller invokes those monitors, and performs a new Bayesian update using the post-action diagnosis vector as the apriori probability distribution and the new monitor outputs. The whole process then repeats until the probability of the null fault hypothesis increases to the specified threshold $(1-\epsilon)$, or the algorithm encounters a fault from which it cannot recover (i.e., operator alert).

CoOL Example. Figure 3(b) illustrates a worst case scenario of trying to diagnose a Value(S1) fault even though the CoOL monitors cannot distinguish between S1 and S2 value faults. When the controller is initially called with HPathMon reporting an alarm, the Bayesian diagnosis implicates Value(HG), Value(S1), and Value(S2). As a result, consider what happens if HG.Restart is the (incorrect) action picked, causing the Value(HG) fault to be removed from further consideration. If the next round of monitor execution causes an alarm in VPathMon instead (as would be the case for round robin load balancing between S1 and S2), the Bayesian update would eliminate h_ϕ from consideration and implicate Value(S1) and Value(S2) equally. Let us imagine that the controller's response is to pick Restart.S2 (also incorrect), thus eliminating Value(S2) from further consideration. Then, any HPathMon or VPathMon alarm in the next monitoring round would negate the null fault, and cast suspicion on Value(S1) alone, causing it to be restarted, and terminating the algorithm. Thus, the recovery algorithm eventually terminates successfully despite making early mistakes due to imprecise diagnosis.

V. Recovery Action Selection

Next, we discuss how actions can be selected based on the current diagnosis vector and the system models.

A. Single Step Lookahead

The recovery example illustrated previously hints at a straightforward way to choose recovery actions - choose the one that fixes at-least one of the most likely faults at the lowest cost. We implement such a strategy in a *Single Step Lookahead (SSLRecover)* recovery controller. To quantify the cost of an action, SSLRecover accepts a cost metric $a.\text{cost}$ as input for each action. In the CoOL example, we use the action duration multiplied by the fraction of user requests flowing through the target component as the cost metric.

Since SSLRecover greedily makes its choice by "looking" only one recovery action ahead, it works only when all faults are fixable by at-least one recovery action in a single step, and can choose only those actions that fix at-least one fault in a single step. Specifically, $\forall h \in \mathcal{H}, \exists a \in \mathcal{A}$ s.t. $a.\text{hEffect}(h) = \text{h}_\phi$, and for an action a to be selectable, $\exists h \in \mathcal{H}$ s.t. $a.\text{hEffect}(h) = \text{h}_\phi$. These restrictions have important consequences; SSLRecover cannot use actions whose outcomes depend on the order in which they are applied. E.g., restarting a process before recovering its corrupt configuration files (using a backup) is likely to fail. More importantly, it also cannot utilize "monitor only" actions that do nothing besides return diagnostic information. Nevertheless, if its conditions are met, it can be shown that SSLRecover will always terminate successfully in a finite number of steps by eliminating fault hypotheses one at a time [9]. However, it cannot optimize cost metrics such as down time or recovery duration that are functions of the entire sequence of actions executed during recovery, and which are important in practice.

B. Multistep Lookahead

To address the shortcomings of a greedy approach, we present a *Multi Step Lookahead (MSLRecover)* recovery controller based on a Partially Observable Markov Decision Process (POMDP) construction that evaluates and optimizes over entire sequences of future recovery actions before choosing an action.

System Model. At the core of the MSLRecover algorithm is a state-based model of the target system, fault hypotheses, and recovery actions which augments the definitions provided in Section III. The model state is a tuple (system state, fault hypothesis). System states $\mathcal{S}$ are defined using a set of state-variables $\mathcal{SV}$, with each state reflecting a unique assignment of values to each state-variable. System state is assumed to be fully observable, and its current value can be read from the system at any time. Fault hypotheses are encoded as a single state variable whose true (but unknown) value reflects the fault currently present in the system. The value of this unobservable state variable is represented using the diagnosis vector $\{\text{P}[h]\}$.

Each recovery action a is represented by a precondition and a state effect in addition to the fault hypothesis effect defined by the $a.\text{hEffect}$ function. The precondition $a.\text{pre}(s)$ is a Boolean-valued function that returns true only if a is allowed in observable state s, while the state effect $a.\text{sEffect}(s) \rightarrow \mathcal{S}$ specifies how a modifies the system state. Finally, $P[m|h](s)$, $a.\text{hEffect}(s,h) \rightarrow \mathcal{H}$ and $a.\mathcal{M}(s) \rightarrow 2^{\mathcal{M}}$ are monitor coverage, fault hypothesis effect, and monitor selection functions augmented to allow their values to be dependent on observable system state s.

A cost metric allows us to specify the cost of actions (as well as inaction) and MSLRecover algorithm aims to minimize the overall costs. The cost metric is defined on a per-step basis (where the word "step" refers to a single action executed by the controller). Cost is defined using a rate cost that is accrued continuously at a rate $\mathbf{c}_r(s,a,h)$ and an impulse cost $\mathbf{c}_i(s,a,h)$ that is accrued instantaneously when the controller chooses action a in model-state (s,h). With these definitions, the single-step cost is computed as $\hat{\mathbf{c}}(s,a,h) = \mathbf{c}_i(s,a,h) + \mathbf{c}_r(s,a,h) \cdot \{a.\bar{t}(s,h) + \text{monitor}.\bar{t}\}$ where $a.\bar{t}(s,h)$ is the time

taken to execute action a, and monitor.$\bar{t}$ is the time taken to execute the monitors. Costs can be negative to model profitable actions.

CoOL Example. For the CoOL system, the state includes the current set of hosts $\mathcal{R}$, components $\mathcal{C}$, monitors $\mathcal{M}$, and a partition that specifies the set of components running on each host r. Making these definitions a part of the state allows recovery actions to change them (e.g., enable/disable hosts or components), but to facilitate comparisons to SSLRecover, we consider only *c*.restart and *r*.reboot actions along with a monitor-only action *Test*. All actions are always enabled (pre always returns true), and they do not have any state effects (sEffect is the identity function). Definitions of monitor coverages, a.hEffect, and $a.\mathcal{M}$ are not system-state-dependent and remain unchanged from Section III. Finally, we seek to minimize the number of user requests impacted by failures, and so the rate cost (assuming an equal mix of voice and HTTP requests) is defined as $\mathbf{c_r}(s,a,h) := \mathbf{1}[h \in \{\text{Down(hC), Crash(DB), Value(DB)}\}] + 0.5 \cdot \mathbf{1}[h \in \{\text{Crash(S1 or S2 or HG or VG), Value(S1 or S2 or HG or VG)}\}] + 0.25 \cdot \mathbf{1}[h \in \{\text{Down(hA), Down(hB)}\}]$. The fraction of impacted requests is 1 while the database or its host is down, 0.5 while either gateway is down (due to the workload mix) or either EMN server is down (due to the load balancing), and 0.25 while HostA or HostB is down (because multiple components are affected).

Optimization Framework. We can now cast the optimal recovery problem in terms of the model elements above. Let $(S_t, \{\text{P}[h]\}_t)$, A_t, and H_t be random variables denoting the model-state, chosen recovery action, and correct fault hypothesis during a particular step t of the recovery controller. If T is the total number of steps needed for recovery, the goal of the MSLRecover algorithm is to choose a sequence of actions $a_1, \ldots, a_T$ such that the mean value of the one-step cost summed over the entire recovery process $\mathbb{E}\left[\sum_{t=1}^{T} \hat{\mathbf{c}}(S_t, A_t, H_t)\right]$ is minimized. Let $V(s, \{\text{P}[h]\})$ represent the minimum "recovery cost-to-go" possible when the system is in state s with diagnosis vector $\{\text{P}[h]\}$; we call it the *mincost* for the current state. The choice of action that leads to the mincost is then the optimal recovery action that can be currently taken. To solve the optimization and calculate mincost, we turn to the framework of Partially Observable Markov Decision Processes (POMDP) with a total-cost criterion.

Briefly, a POMDP [11] is a system (S, A, O, p, q, r) that can execute a variety of actions A, but whose states S are not directly observable except through emitted observations O. The transition function $p(s, a, s')$ specifies the probability of the system transitioning from state s to state s' when action a is applied, while the observation function $q(o, a, s)$ specifies the probability with which observation o is produced when the system transitions to state s as a result of action a. Finally, the reward $r(s, a)$ defines the reward obtained by the system when action a is chosen in state s. It is known that the maximum value of the mean reward or "value" V^* that can be accumulated by a POMDP (with restrictions to ensure finiteness) is solely a function of the state-occupancy-probability distribution $\pi = \{\text{P}[s] | s \in S\}$, which is also called the belief state. This optimal value is given by a Bellman dynamic programming recursion: $V^*(\pi) = \max_{a \in A} \sum_{s \in S} \pi[s] \cdot r(s,a) + \sum_{o \in O} \gamma(o|\pi, a, s) V^*(T(\pi, a, o))$. Here, $\gamma(o|\pi, a, s) = q(o, a, s) \sum_{s' \in \mathcal{S}} p(s', a, s) \pi(s')$ is the probability that observation o is produced due to a transition to state s as a result of action a being executed in belief state π, while $T(\pi, a, o)$ represents the new belief state obtained using Bayes rule when action a is executed in the current belief state π and observation o is seen as a result. It's unnormalized value for state s is given by $T(\pi, a, o)[s] = q(o|a, s) \sum_{s' \in \mathcal{S}} p(s|s', a) \pi(s')$.

Optimal Action Selection. The POMDP formulation can be adapted for the recovery problem by considering model states $\{(s, h)\}$ as the POMDP states, and monitor outputs $o_{\mathcal{M}}$ as the observations O, and minimizing the cost instead of maximizing reward. This would lead to a belief state defined over the entire model-state-space, i.e., $\pi = \{\text{P}[(s, h)] | s \in \mathcal{S}, h \in \mathcal{H}\}$. To enable a more compact representation and smaller state-space, we use the observation that the current system state s is fully observable and thus the probability $P[(s', h)] = 0, \forall s' \neq s$. Therefore, the belief state can simply be represented as $(s, \{P[h] | h \in \mathcal{H}\})$, i.e., a pair consisting of the current system state and the diagnosis vector. Then, the transition function is given by $p((s, h), a, (s', h')) = 1$ if $a.\text{pre}(s) = \text{true}, a.\text{sEffect}(s) = s', a.\text{hEffect}(s, h) = h'$, and 0 otherwise. The observation function is computed using Equation 2 from Section IV as $q(o, a, s) = P[o_{a.\mathcal{M}} | (s, h)]$ and finally, the single-step cost function can be used unmodified. The resulting equation for the mincost as a function of the system state s and diagnosis vector $\{\text{P}[h]\}$ is given by Equation 3.

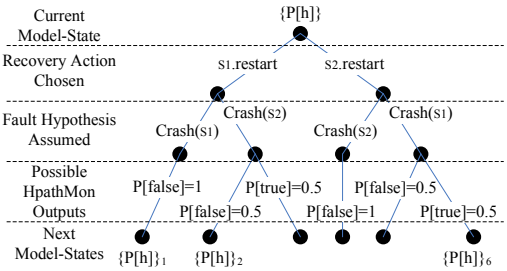


Fig. 4. Example Step of the MSLRecover Computation Tree

The easiest way to understand Equation 3 is through a sample single-step of the recursion tree as shown in Figure 4. For each possible combination of fault-hypothesis h and chosen recovery action a, the equation considers the possible next model-states the system can transition into. It does so by computing the set $\mathcal{O}_h$ of possible monitor output combinations that could possibly be generated after a was executed with h present in the system. For each such possible monitor output combination $o_{a.\mathcal{M}}$, the next system-state is just $a.\text{sEffect}(s, h)$. To compute the next diagnosis vector corresponding to $o_{a.\mathcal{M}}$, we first compute how execution of action a affects the fault

$$V(s,\{\mathrm{P}[h]\}) = \min_{\substack{a\in A\\ a.\mathbf{pre}(s)}} \sum_{h\in\mathcal{H}} \mathrm{P}[h]\Big\{\hat{\mathbf{c}}(s,a,h) + \sum_{o_{a.\mathcal{M}}\in\mathcal{O}_h} \mathrm{P}[o|s,a,h]V(a.\mathbf{sEffect}(s), \textsc{Bayes}(\{\mathrm{P}[h|s,a]\}, o_{a.\mathcal{M}}))\Big\} \quad (3)$$

hypothesis probabilities, using the following equation:

$$\mathrm{P}[h|s,a] = \sum_{h'\in\mathcal{H}} \mathrm{P}[h']\mathbf{1}[a.\mathbf{hEffect}(s,h') = h] \quad (4)$$

Then, simply calling the Bayesian update from Equation 1 with $\mathrm{P}[h|s,a]$ and $o_{a.\mathcal{M}}$ as inputs results in the next diagnosis vector. The corresponding transition probability is the probability that monitor outputs $o_{a.\mathcal{M}}$ are generated after a is executed with h in the system. This probability is given by (with $\mathrm{P}[o_{a.\mathcal{M}}|(s,h)]$ computed using Equation 2):

$$\mathrm{P}[o|s,a,h] = \mathrm{P}[o_{a.\mathcal{M}}|(a.\mathbf{sEffect}(s), a.\mathbf{hEffect}(s,h))] \quad (5)$$

The mincost of a state if action a is executed with fault hypothesis h in the system is then given by the sum of the single-step cost and the expected value of the mincost of the next model-state (computed as a sum over the set of next states weighted by their transition probability). Averaging over $\mathcal{H}$ then results in the mincost that results from choosing action a. The action that minimizes the mincost is the action chosen. Figure 4 shows this process in the context of the CoOL system with only two fault hypotheses (i.e., Crash(S1) and Crash(S2)). The two recovery actions being considered are S1.restart() and S2.restart(). If the correct action is chosen, then the only possible monitor output is false. However, if an incorrect action is chosen, the monitor will detect the fault only with probability 0.5 (its coverage), and two outputs are possible.

One issue with Equation 3 is the size of set $\mathcal{O}_h$. In the worst case, the number of possible monitor output combinations is exponential in the number of monitors. However, in practice, that is usually not a problem, because if a monitor either does not detect a fault hypothesis at all, or detects it with certainty (coverage 1), it has a single output for that fault hypothesis, and does not cause addition of elements to $\mathcal{O}_h$. In most practical systems, that is what happens since most monitors either detect only a few fault hypotheses or detect their targeted faults with certainty. However, a systematic study of scalability is outside the scope of this paper, and remains an avenue for future work.

Finite-Depth Exploration If Equation 3 is solved exactly for the choice of the best action, the recovery is guaranteed to be optimal. However, because the model-state space is infinite (all the possible values of $\{\mathrm{P}[h]\}$), the recursion is notoriously difficult to solve. We circumvent the difficulty by sacrificing the quest for provable optimality and expanding the recursion only up to a finite depth *maxdepth* (or until a state is reached where the system has recovered and $\mathrm{P}[\mathrm{h}_\phi] > 1\text{-}\epsilon$). That ensures that only a finite number of values of $\{\mathrm{P}[h]\}$ are needed in the solution. However, that also means that when the recursion is terminated because the maximum depth was reached, a heuristic must be used to represent the remaining cost of recovery at the point of termination.

The value of the heuristic can have an important effect on the efficiency of the recovery actions generated by the algorithm. In our implementation, we have used the following heuristic, which gives good results (as witnessed by the fault injection experiments in Section VI): $\mathrm{V}(s,\{\mathrm{P}[h]\}) = (1 - \mathrm{P}[\mathrm{h}_\phi])\cdot \max_{a\in\mathcal{A}} a.\bar{t}$. Essentially, the heuristic penalizes actions that do not move probability mass to h_ϕ. We also experimented with another heuristic that favored short recovery sequences (in the number of actions) by assigning a constant large cost to every V(s,{P[h]}). However, this heuristic caused the algorithm to behave too aggressively by picking expensive recovery actions that ensured recovery within less than the lookahead horizon (maxdepth). They prevented the algorithm from outperforming SSLRecover. While outside the scope of this paper, we have also developed bounds [9] instead of heuristics to ensure strong guarantees on recovery cost. In conclusion, we would like to remark that even though finite-depth recursion can lead to non-optimal solutions, we believe that most practical faults can be recovered from using only very few recovery steps, and MSLRecover does well when recovering from such faults even when a very small recursion depth is used.

We have implemented both recovery algorithms in C++ as part of a toolbox for model-driven adaptation and recovery [9]. The SSLRecover algorithm accepts specifications of fault hypotheses, monitors, coverage probabilities, and recovery actions via a simple textual description language. However, MSLRecover requires not only a description of relevant system state, but the reward and recovery action specifications (e.g., the pre, hEffect, sEffect, and $\bar{t}$ functions) required can be complex functions of the state. Therefore, our implementation allows the user to specify the state, action, and reward specifications using arbitrary C++ code embedded in a higher-level model description language. The specifications are automatically converted into C++ code that, when compiled together with the recovery algorithm library, provides an implementation of the recovery controller. The controller requires callbacks to be implemented in the system to execute recovery actions and system monitors.

VI. Simulation Results

To validate the recovery algorithms and measure the system availability provided by automatic recovery over a significant period of time, we use the algorithms on a simulated CoOL system with simulated faults. To compensate for the lack of SSLRecover's ability to decide when more monitoring is needed, we deploy an additional version for comparison in which monitors are invoked and diagnosis is performed twice before invoking a recovery action. The repeated invocation of the monitors and the Bayesian estimation is expected to help with the probabilistic nature of the HPathMon and VPathMon end-to-end monitors by better diagnosing the failure before action is taken.

We have implemented a simulation harness that interfaces with the SSL and MSL controllers, but simulates the rest of the system to provide the controllers with stimuli similar to those that would be generated in an actual EMN deployment. The simulator injects faults into the system one at a time using a

Poisson arrival process with a specified rate λ_f, with the fault chosen from the set of available fault types according to a uniform distribution. Monitors are invoked once every minute (i.e., faults are not always detected instantaneously), and take 5 seconds to execute. If the monitors detect an exceptional condition, the recovery algorithm is invoked. The recovery and monitoring actions generated by the recovery algorithms are simulated. However, the recovery algorithm is not simulated, and runs the implementation in real time.

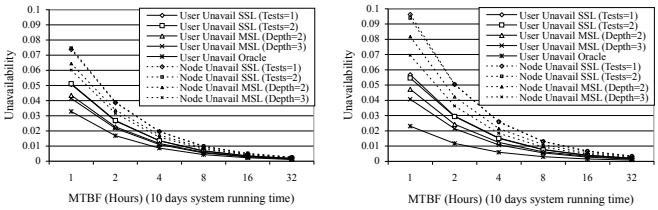
To recall, the CoOL model contains 5 components running on 3 hosts, 13 fault hypotheses (h.Down, c.Crash, and c.Value), 7 monitors (Ping(c), HPathMon, and VPathMon), and 9 actions, (c.Restart, h.Reboot, and a monitor only "Test" action for MSLRecover). The Reboot actions took 5 minutes regardless of which host was being rebooted, while the Restart actions took 1 minute for the HTTP gateway, 2 minutes for the voice gateway and EMN servers, and 4 minutes for the DB. Finally, the system was exercised by a workload consisting of 80% HTTP traffic and 20% voice traffic. This was done to allow the different gateways to have different importance regarding user-perceived availability (fraction of successful user requests), and test the ability of the recovery algorithms to prioritize recovery based on component importance.

A. Availability under Fault Injection

Four different configurations were considered: the SSLRecover algorithm with the number of test and diagnosis rounds to invoke before choosing a recovery action (*maxtests*) set to 1 or 2, and the MSLRecover algorithm with maximum lookahead depth (*maxdepth*) set to 2 or 3. Finally, we ran simulations using an unrealizable "recovery oracle." In the recovery oracle runs, the recovery algorithm is assumed to have perfect knowledge of what fault has occurred, and hence can always choose the best recovery action (as predetermined by a human). The oracle represents the best possible performance any algorithm could have given the types of faults and recovery actions available to the system.

The experiments were conducted on hosts with Athlon XP 2400 processors. For each simulation run, the experiment was terminated at the end of 10 days of simulation time. 100 independent runs were performed for each configuration. To eliminate transient effects, measurements were started after the controller signalled that it had finished initialization. Each of the five configurations was tested in two different scenarios. In one, all thirteen types of faults were injected and in the other, only Value faults were injected, (although the recovery algorithms did not know that). Value faults were considered separately because they cannot be detected with good localization and are a challenge for automatic recovery.

The graphs in Figure 5 show both the user-perceived unavailability, i.e., the fraction of requests that traverse a faulty component, and node unavailability, i.e., the fraction of time some component is faulty, as a function of the MTBF when all types of faults and only value faults were injected. Each value in the graph is a mean over 100 runs, and in all cases, the 99% confidence interval is less than +/-5% of the mean. First, it can be seen that system unavailability remains more or less the same as the value of maxtests is increased for the SSLRecover algorithm, but decreases as the value of maxdepth is increased for MSLRecover. Second, the system availability using the MSLRecover algorithm is better than the availability using the SSLRecover, and this difference is more pronounced for Value faults which, unlike crash faults, may require multiple steps to effect recovery, and a longer lookahead has greater benefit. Third, the user perceived unavailability is much higher than node unavailability, in part due to the fact that the recovery algorithms can prioritize recovery actions based on the importance of the target components. Finally, although the algorithms fare reasonably well with respect to the oracle benchmark, there is still room for improvement in the type of monitoring available to this system. Moreover, the difference is more pronounced for value faults because of their poor diagnosability.



(a) All Faults (b) Value Faults only

Fig. 5. Unavailability as a function of MTBF

TABLE II
FAULT INJECTION RESULTS (VALUES ARE PER-FAULT AVERAGES)

Faults All	Cost	Down Time	Fault Time	Alg. Time	Num Act.	Num Mon.
SSL1	199.79	290.68s	226.33s	0.08ms	1.837	2.89
SSL2	199.61	291.69s	227.02s	0.12ms	1.802	4.653
MSL2	169.50	251.87s	205.82s	13.2ms	1.590	2.868
MSL3	160.56	233.59s	197.68s	221ms	1.428	3.079
Oracle	121.15	179.77s	n/a	n/a	1.000	0.000
Value						
SSL1	229.58	389.28s	220.98s	0.11ms	3.000	4.13
SSL2	222.41	381.93s	214.38s	0.02ms	2.901	6.938
MSL2	186.44	323.40s	203.64s	33.6ms	2.554	4.260
MSL3	161.94	274.13s	181.00s	490ms	2.130	4.837
Oracle	84.40	132.00s	n/a	n/a	1.000	0.000

B. Recovery Benchmarks

We present more detailed measurements of the algorithms in Table II. Each measurement is a mean across all injected faults, a total of more than 40,000 for each controller. All 99% confidence intervals are smaller than +/-5% of the means. In addition to indicating the total number of faults injected for each configuration, the table specifies many per-fault metrics. The cost metric specifies the cost of recovery per fault (system execution time $\times$ (1-availability)/number of faults). The metric essentially weighs the down time for each component by the importance of the component. The down time metric represents the unweighed amount of time (per fault) when some component in the system is unavailable either because of a fault, or because the recovery algorithm is restarting or rebooting it. The residual time metric specifies the average amount of time a fault remains in the system. The algorithm time represents the time (wall-clock) spent in executing the MSLRecover and SSLRecover algorithms (i.e., not including the cost of monitoring and repair actions). Finally, the actions

TABLE III
ERRORS INTRODUCED IN MONITOR COVERAGE.

Fault Hyp.	Null	Null	¬Null	¬Null	¬Null
Orig. Cvg.	0	¬0	0	1	¬(0 or 1)
Faulty Cvg.	0.2	Rand.	1	0	Rand.

and monitor calls entries are the average number of recovery actions and test actions executed by the recovery algorithms per fault recovery.

Several observations can be made from Table II. First, the fault residual time is actually smaller than the down time because of value faults that have poor diagnosability. Even after recovering from a fault, the controller does not know for sure that the fault was really fixed due to the probabilistic nature of the monitors. Therefore, it may still execute additional recovery actions to bring the probability of the null fault hypothesis (h_ϕ) to within the target of $1-\epsilon$ (ϵ was set to 0.001 in our experiments). Increasing the value of ϵ will reduce the down time and cost of recovery (as we confirmed in our experiments), but it could also increase the likelihood that the algorithm will terminate with the fault still present in the system.

The second observation is related to the average number of actions vs. the number of monitor calls for the two algorithms. Notice that increasing the maxtests parameter (for SSLRecover) or maxdepth (for MSLRecover) increases the efficiency of the respective algorithms in the sense of fewer average actions per fault. However, the MSLRecover gains a much larger benefit from a far lesser increase in the number of monitoring actions. The reasons are both its ability to initiate additional testing only when needed, and a longer lookahead that gives it the ability to better see the future benefits of additional testing. In contrast, increasing maxtests for SSLRecover increases the testing for *all* types of faults, even when it may not be needed (e.g., for the easily diagnosable crash failures).

Finally, we also point out the execution times of the two algorithms. SSLRecover has shorter execution time because of its polynomial complexity with respect to the number of fault hypotheses and actions. Nevertheless, provided that the lookahead for MSLRecover is kept small, both algorithms are fast enough for real systems. Techniques that speed up the MSLRecover algorithm and allow an increased lookahead are topics for future research.

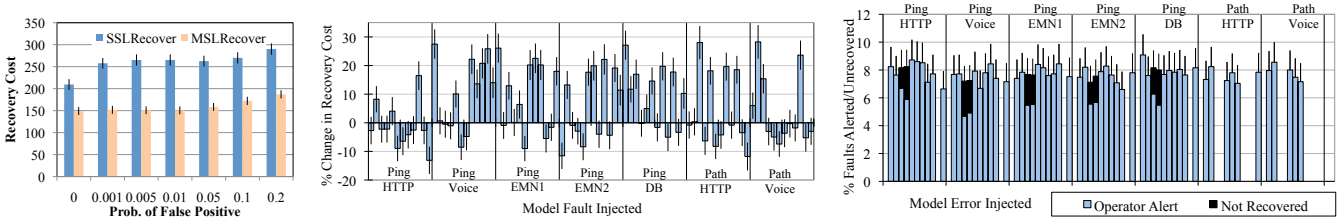
C. Sensitivity to Errors

The final set of simulation results show the effect of monitor noise (i.e., false positives) and model inaccuracies on the quality of recovery. To examine the impact of noise, we conducted fault injection experiments with increasing values of the false positive probabilities P_{fp} for all monitors. In each experiment, we replace all 0 monitor coverage values with P_{fp} and conduct 1000 fault injections with both SSLRecover (maxtests=1) and MSLRecover (depth=2) controllers. The results are presented in Figure 6(a) with 99% confidence intervals and show that as the false positive rate increases, recovery takes longer for both controllers. For the SSLRecover algorithm, the impact is most severe when going from 0 to non-zero P_{fp}, while MSLRecover can cope very well with small increases in false positives. Impacts of very high false positive rates are also more severe for SSLRecover (38% recovery cost increase when going from 0 to a 20% false positive rate) than for the MSLRecover algorithm (only a 25% increase). These results can be attributed to MSLRecover's ability to invoke additional monitoring when needed, and thus gracefully deal with false positives.

To examine the impact of errors in monitor models on the MSLRecover algorithm, we systematically introduced errors into monitor coverages in the CoOL system model. The erroneous models were used with the MSLRecover algorithm (depth=3), and using random action leaf heuristics that were iteratively refined using 10 bootstrap iterations of the Vertex refinement heuristic as described in [9]. In each set of experiments, we picked a single monitor m, a single fault hypothesis h, introduced an error in the corresponding monitor coverage $P[m|h]$, and observed the recovery performance of the resulting controller for 2500 fault injections into the simulated system.

The errors we introduced are shown in Table III. In the table, the first row shows the type of fault hypothesis whose monitor coverage probability was modified, while the second and third rows show the original and modified values of the coverage probability respectively. Specifically, both errors of omission in which a monitor model omits a fault class actually detected by the monitor (i.e., setting monitor coverage of 1 to 0) and errors of commission where a monitor model indicates that the monitor detects a fault that it does not detect in reality (i.e., setting a monitor coverage of 0 to 1) were introduced. Errors that change a null fault coverage of 0 to 0.2 are equivalent to falsely believing that a monitor generates false positives 20% of the time when in fact it does not generate any. Monitors having a null fault coverage of 1 (corresponding to a monitor that always generated false positives) were discarded as unrealistic. For monitor coverages that are neither 0 nor 1, errors typically reflect inaccuracies in probability estimation rather than structural errors such as omission or commission. Hence, to avoid an unrealistic choice of a single error value, for every experiment we set the coverage to a different value chosen uniformly at random from the range $(0, 1)$. The mean across all experiments reflects average results across a variety of probability estimation errors.

Figure 6(b) shows the percentage increase in the mean recovery cost over a controller using the correct model, as calculated over 2500 fault injections along with 99% confidence intervals. In the graph, the x axis indicates the monitor and fault hypothesis whose coverage was altered. For each monitor label in the graph, results are shown, in order, for the Null fault hypothesis, value faults in the HTTP gateway, Voice gateway, EMN1 server, EMN2 server, and database server, crash failures in the HTTP gateway, Voice gateway, EMN1 server, EMN2 server, and database server, and finally, crash failures in Hosts A and B. As can be seen, the change in mean cost of recovery for single model errors changes unpredictably according to the particular error that was injected, but is within 30% of the recovery cost obtainable via a correct model. Surprisingly, for some model errors, a controller with an incorrect model actually performed better than a controller with a correct



(a) Effect of false positives (b) Change in recovery cost due to errors (c) % of unrecovered faults & alerts due to errors

Fig. 6. Impact of noise and error in monitor coverage

model. Further investigation revealed that the lower recovery costs in some cases were due to the fact that for several faults, the controller was terminating early without recovering from the fault present in the system, and thus giving the impression that recovery was quicker.

We computed two metrics to quantify the early termination: the percentage of faults which the controller did not recover from because it falsely believed the recovery to be complete and the percentage of faults the controller was not able to recover from, but alerted the operator to initiate manual recovery. The results are shown in Figure 6(c), also with 99% confidence intervals. As can be seen from the graph, there is a substantial fraction of faults, up to 9% in some cases, from which the controller could not recover whenever coverages were tampered with. Fortunately, in most of these situations, the controller was also able to quickly detect that something was wrong and alert the operator, thus leading to the "reduced" recovery costs. This is because of the 1 or 0 values of many of the coverages in the CoOL system model. When these probabilities were inverted due to the model errors introduced, they led to situations in which the set of monitor observations being generated had a zero probability of occurring according to the model (computed using Equation 2), thus leading to an operator alert.

The only faults for which the controllers silently terminated recovery early were Value faults in either of the EMN servers. We discovered that if the monitor coverage for one of the two EMN server Value faults was increased to a high enough random value from its original value of 0.5, the controller mistakenly estimated that the fault had been recovered from with a high probability after one or two rounds of monitoring, and terminated. In practice, even this issue is not a significant problem because the monitors would eventually detect the fault again and initiate a new round of recovery. Moreover, these situations only occurred when the modified coverage was significantly different from its true value.

VII. Selective Restart in EJB Systems

To demonstrate MSLRecover's capabilities to choose recovery as well as information gathering diagnostic actions, we conducted an experimental case study in which it was used to adaptively choose test probes and perform selective restarts of components in web services written using the J2EE Enterprise Java Beans (EJB) framework [12]. Similar to the CoOL system, EJB applications typically follow a 3-tier model with a front-end tier of Web servers or Java Servlet containers (e.g., Apache Tomcat), a middle tier of EJB application servers that implement the business logic, and a back-end database tier that provides a persistent store for the system. "JavaBeans" are software modules hosted in J2EE application servers that represent business entities and implement business logic.

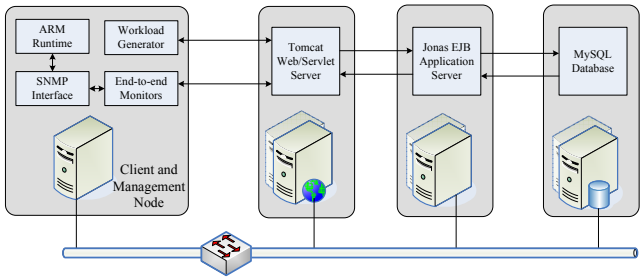
A. The RUBiS EJB Application Models

The particular EJB application we used is RUBiS [6], an open-source auction site prototype modeled after eBay that implements functionality such as selling, browsing, and bidding. RUBiS includes a client emulator for benchmarking. The RUBiS client emulates user behavior for various workload patterns (visitor, buyer, or seller) and collects statistics (number of requests completed, response time, and number of failed requests). The emulator works by generating sessions, i.e., a sequence of interactions from the same customer, for multiple simulated customers.

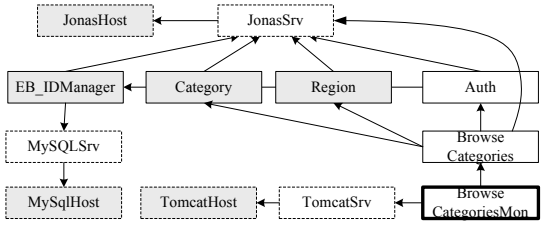
We consider a typical setup using the session-facade version of RUBiS as shown in Figure 7(a), where the functionality is divided into three types of objects: servlets that reside in an Apache Tomcat Web container, and session and entity EJBs that reside in the JOnAS EJB application server [13]. The session beans implement the core functionality of the system while the entity beans represent persistent objects such as users, items, bids, and comments. There are 16 stateless servlets, one for each type of user request, which invoke session EJB methods and convert the output to HTML to return to the user. There are 17 session beans in JOnAS server, one corresponding to each servlet and an additional one for authentication, while there are 10 entity beans shared by multiple session beans. To allow beans to be restarted individually, each bean was hosted in a separate container. Finally, a MySQL database was used to provide persistence for the Entity beans. In our testbed setup, the three servers were located on a separate physical hosts.

Fault Model. We consider two different types of faults, temporary bean faults and temporary server faults. The first are faults that occur in a specific bean type (e.g., null pointer accesses, errors in remote object invocation, or other application-specific exceptions thrown by the bean itself), and can be fixed by restarting the container in which the bean is hosted using the ResetBean action. The time taken for redeploying a container is very small—we measured it as less than 800msec. Therefore, the execution time for the ResetBean action is set to 800msec in the model.

The second type of faults are temporary server faults which occur in the Tomcat, JOnAS, and MySQL servers. These faults affect all EJB containers and beans if they occur in the JOnAS server, and all the servlets and static Web pages if they occur in the Tomcat server. They include not only server crash



(a) Experimental setup for RUBiS



(b) Monitor coverage graph for BrowseCategoriesMon

Fig. 7. RUBiS experimental setup and monitor models

failures, but also other faults that are not fail-silent, e.g., faults due to insufficient memory. The ResetServer action restores a temporary server fault. For the JOnAS server, the ResetServer action also implicitly clears the faults in the individual beans. Restarting JOnAS was a relatively expensive operation, with restarts taking up to 30 seconds, while restarting Tomcat took up to 8 seconds and restarting MySQL took up to 15 seconds. We used these measurements to set the action times for the corresponding restart actions in the model. Although these times are comparatively small, restarting servers has costs other than just temporary loss of availability since restarts also terminate the ongoing sessions of users accessing the service.

Monitor Models We use both SNMP-monitors and end-to-end monitors in our RUBiS deployment. TomcatMon and JOnASMon monitor Tomcat and JOnAS, respectively, by querying the built-in Java Runtime SNMP engine for the jvmThreadCount.0 variable, which returns the current number of threads running within the JRE. The monitors flag an alarm if the variable cannot be accessed, or if it becomes 0. Thus, these monitors monitor both the server processes and the hosts on which they execute. End-to-end monitors (probes) test the application components (beans) using emulated client requests for each transaction type on a dummy test account. The probes are generated using the cURL tool, and the system's response is checked for error strings. Due to the large number of transaction types, by default only the SNMP monitors are always invoked after each recovery action. To allow the controller to explicitly choose which probes would provide the most relevant information, we include separate "monitor only" actions corresponding to each transaction. The $a.\mathcal{M}$ set for these actions includes the SNMP monitors and the probe for a single transaction, and they do nothing else except execute the probe. These monitor only actions precluded the use of a SSLRecover controller in these experiments .

Monitor coverage is specified through coverage graphs, an example of which is shown in Figure 7(b). In the figure, the solid white and gray boxes represent fault hypotheses in session and entity beans respectively, while the dashed outline white and gray boxes represent faults in server processes and hosts. Each session bean is tested by one end-to-end monitor and as a result, most entity beans are tested by multiple end-to-end monitors. The monitor coverage graph was constructed by tracking the request flows for each request type based on the deployment descriptor that is provided with every EJB application. However, they could also have been extracted either by static analysis of the code, or via fault injection ([14]). Monitors have a coverage of 1.0 for each of the fault hypotheses in its monitor coverage graph. It might be argued that such perfect coverage is unrealistic because faults may not always effect the entire user population (e.g., faults might affect only requests on a subset of the database tables or users). However, that is a limitation of the monitoring technique rather than the models. A fault will either affect the test account or not, and thus coverages other than 1.0 would be of little use unless the fault is an intermittent one. More complete coverage could be achieved by putting the error detection code within a client-side script on the HTML pages returned to the user so that all users can act as monitors and participate in error reporting.

Finally, a fictitious "user" monitor is defined whose coverage graph reflects the call-flow produced by all the user transactions combined (with the routing probabilities given by workload mix). The cost metric for the system is then defined as the probability that the user monitor "detects" either a fault or any (correct or incorrect) restart action being executed by the controller, i.e., the fraction of users affected by the fault or recovery action. While this cost model does not factor the number of sessions reset by an incorrect restart during the optimization, the experimental measurements of errors do take such resets into account.

B. Fault Injection Results

We conducted fault injection experiments by injecting component crashes at the server and bean levels. Server crashes were implemented by sending a KILL signal to the appropriate process, while a bean crash was emulated by undeploying the container in which the bean resided using the JOnAS admin command line tool. As already noted, each bean was deployed in a separate container to facilitate easy redeployment.

A browse-only workload mix consisting of 240 users was generated using the RUBiS client.The monitors were invoked once every 5 seconds. Each experiment lasted approximately 6 minutes, with a 30-second startup phase during which the request rate was slowly ramped up, a 10-second shutdown phase to allow pending clients to finish, and a 5-minute measurement phase over which measurements of the request rate, and the number of errors seen by clients were made. Halfway through the measurement phase, a single fault was injected in each experiment, with 10 experiments conducted for each fault type. In the experiments, the system workload was low enough that we did not observe any false positives due to packets or requests being dropped.

Table IV shows the results obtained from injecting faults in each of the servers, and faults in three of the beans.

TABLE IV
NUMBER OF ERRORS UNDER DIFFERENT RECOVERY POLICIES

Fault Injected	Recovery Policy		
	Ping Only	Full Reset	MSLRecover
Tomcat Crash	284.12	305.76	293.85
JOnAS Crash	1299.54	1292.94	1304.8
MySQL Crash	4887.22	2347.8	591.68
DB Connection Drop	4880.67	2238.96	230.72
Item Bean Crash	1249.47	2322.15	62.60
Region Bean Crash	956.25	2328.74	52.06

Specifically, it shows the number of client sessions that were terminated because of errors due to the injected fault, timeouts due to unavailability, and connection drops due to component restarts. Two other common recovery policies were used for comparison. Ping Only is a predefined policy that relies only on the outputs of the SNMP-based ping monitors. When a ping monitor indicates a failure, that particular server is restarted. The Full Reset is a policy that uses the outputs of both the SNMP-based and the end-to-end monitors. If an SNMP-based monitor reports an alarm, the corresponding server is reset. However, if the end-to-end monitors report a problem, the entire system is reset, because the policy cannot determine in which component the fault might lie. The MSLRecover policy used had a lookahead depth of 3.

The results show that when the Tomcat server and the JOnAS server crashed, all three policies performed similarly. This was expected because all three policies can perfectly detect the server crash via the ping monitors and take corrective action. The number of requests dropped when restarting JOnAS is greater than the number of requests dropped when restarting Tomcat simply because JOnAS takes longer to restart. However, for the other fault injections, results differ. Since we could not find an SNMP module for monitoring MySQL, we decided to leave it unmonitored via pings. Therefore, when the MySQL server crashes, the Ping Only policy performs the worst because it does not have the ability to detect the server crash. Both the Full Reset and MSLRecover policies detect the crash through error messages returned in the EJB output and propagated to the monitor. However, the Full Reset policy executes a full system restart, which takes a long time because it is done serially, while MSLRecover uses the more time-efficient alternative of first trying a reset of the EB_IDManager bean followed by a restart of the MySQL server.

When bean failures are injected—including the DB connection drop since it was achieved by undeploying the IDManager bean—the situation is similar. The Ping Only policy does not detect the failure, while both the Full Reset and Selective Reset policies do because of the error messages in the affected requests. The Selective Reset policy is able to isolate the problem in the first round of monitoring and chooses to restart the offending bean—which is a quick operation—while the Full Reset policy restarts the entire system. For Item and Region bean crashes, we were surprised to see fewer errors for the Ping Only policy as compared to the Full Reset policy. The reason turns out to be the experiment run time. Since the experiment terminates approximately two minutes after the fault is injected, it is cheaper just to let the requests that are dependent on the failed bean fail (i.e., the Ping Only policy) than to reset the whole system and ensure that all requests fail in the reset period. We also observed that the number of errors in the case of bean restarts by MSLRecover is larger than would be expected based on the bean restart time. The reason is that, in practice, the monitoring interval (i.e., the time spent before an error is detected) dominates the time required for bean restart, and limits how quickly recovery is initiated.

The results demonstrate that our recovery controller can work for real systems and provides benefits that are not possible through traditional monitoring techniques and static recovery policies. Additionally, we also note that in the experiments presented, no monitor false positives were present. When false positives do occur, the results of a policy such as Full Reset can deteriorate rapidly, as false positives may cause it to invoke expensive recovery actions indiscriminately.

VIII. RELATED AND FUTURE WORK

The issue of detecting and diagnosing failures, and doing something about them has been around as long as computers. Diagnosis in multi-component systems was first formalized using the *system diagnosis* problem in [15]. The basic approach has been subsequently extended in many ways (e.g., to include probabilistic test results [16]). More closely related to our work is *sequential diagnosis* [17], within which the system is repaired incrementally, one repair at a time. The goal is to obtain a *correct sequential diagnosis* in which no correct processor is replaced. Rather than just allow for monitors that can be faulty or correct, we assume that monitor outputs can misrepresent reality in many different ways that can be quantified through the coverage models. Therefore, we can treat monitors with differing specificity and accuracy in a uniform manner. Similarly, we also allow recovery actions with varying effects. In our work, because of limited monitor coverage, it is often inevitable that correct components may be acted upon. Therefore, we optimize the recovery cost rather than strive for strict correctness.

The authors of [18] propose a system diagnosis technique which factors error propagation from one node to another, and tracks message causality in order to trace back to the origin of the fault. In our work, propagation of errors can be encoded in the monitor coverage models. E.g., path monitors model propagation of errors from their source to the tested interface of the system. Coverage models could use message causality graphs instead, in order to account for faults that are known to propagate.

Also related is diagnosis using Bayesian and probabilistic models. The authors of [10] use Bayesian inference for comparison-based system diagnosis to deal with incomplete test coverage, unknown numbers of faulty units, and non-permanent faults. However, they only consider one type of test, and thus does not need to address monitor coverage. In [25], the authors propose a general diagnosis framework using Hidden Markov Models (HMMs), which allows for deviation detection mechanisms with varying coverages, and uses it to diagnose permanent, transient, and internal faults. In [26], the authors use HMMs to distinguish between different failure and attack modes based on coverage models of the failure

modes. [27] shows how to use Information theoretic selection to perform diagnosis based on probes that are very similar to our path monitors. Sherlock [28] provides probabilistic diagnosis for multitier systems by using Bayesian inference on dependency graphs constructed automatically via network sniffing along with user perceived end-to-end behavior. Analogous to our approach of optimizing sequences of recovery actions, *Sequential testing* (e.g. [29]), deals with choosing an optimal sequence of tests in order to perform diagnosis. However, our work is different from all these *diagnose-only* approaches because accurate diagnosis is important only to the extent that it promotes efficient recovery; if there exists a recovery action that cheaply fixes multiple suspected faults, our approach will use it instead of trying to further improve the diagnosis.

Unlike reactive methods that are invoked after a failure has occurred, proactive techniques eschew diagnosis altogether, and periodically invoke recovery actions to restore the system into a known clean state. The concept was introduced as "software rejuvenation" in [1] and considered periodic component restarts to prevent crashes. Subsequent authors, e.g. [30]–[33] have extended the concepts to additional fault models (e.g., Byzantine faults) and recovery actions (e.g., cryptographic rekeying). While rejuvenation is very useful in reducing the occurrence of failures, it cannot eliminate them completely, and thus must be used in addition to reactive approaches.

The question of what automatic recovery actions are available for multi-tier distributed systems has also been studied at depth. In practice, automatic restarting of components is commonly used. Recovery through more selective forms restart, i.e., the "micro-reboots" has been proposed by [2]. They are applied in a strict hierarchical fashion starting from smaller subsystems to larger systems. Fail-over of subsystems is commonly available in most major commercial servers. Rx [34] proposes techniques to recover from software bugs by restarting the application in a different environment designed to mask or remove the bug. However, when used in distributed systems with multiple types of monitors, diagnosis, localization, and cost-benefit tradeoffs are still needed to determine when these techniques should be applied. In that sense, that work is complementary to the contributions of this paper.

There has been some work applying Markov decision theory to deal with cost-benefit tradeoffs when choosing recovery actions. Authors of both [35] and [36] look at the problem of when to invoke a repair process to optimize cost and availability metrics. In both cases, the "optimal policies" that specify when repair is to be invoked as a function of system state are computed via Markov Decision process models of the system. However, both assume full knowledge of system state including complete knowledge of what components have failed, and thus have no notion of monitors or fault diagnosis.

Finally, there has been work on learning repair strategies by direct historical observation. Instance-based-learning is proposed by [37] in which faults and repair actions are injected into the system during a learning phase, and their observed effects are used to learn a *repair policy* that chooses test and repair actions directly as a function of the outcomes of already tried tests. The authors of [38] use Case-based-reasoning (CBR) to choose repair actions using a historical database of (monitor output, repair action) pairs that have been successful in the past. The technique allows for approximate matching to deal with faults that have not been seen before. Because both techniques are based on learning from past observations, they require significant history that can be invalidated if the system structure/configuration changes. Further, because both techniques map monitor outputs directly to actions, adding new monitors or actions requires learning their interactions with existing monitors/actions from scratch.

In contrast, we explicitly separate monitor models from recovery actions through the intermediate construct of fault hypotheses. Thus, new monitors or actions can be added without modifying existing models. Furthermore, explicit models allow the flexibility of being learned, or constructed explicitly when domain information is easily available, as is often the case for monitors such as path monitors, whose coverage values are dependent on easily extractable system call graphs. Finally, we allow non-deterministic and noisy monitors, which can significantly impact the efficiency of learning-based systems.

Future Work

Finally, opportunities for extending our framework lie both on the modeling and systems side.

Modeling Limitations. One of the limitations of our models is that they do not allow for transient faults. Only recovery actions can change fault hypothesis values. Additionally, we consider only one fault hypothesis at a time despite the fact that large systems typically have many failed components at any time. This restriction is not as fundamental as it seems. In a system, there may be many known failures at any given time, but only new undiagnosed faults are of concern to the recovery engine. Typically, older faults will have been diagnosed and recovered or disabled pending repair. Moreover, common mode failures are handled by encoding them using composite fault hypotheses. So, only faults that concurrently and independently occur in the window between occurrence of a fault and its recovery by the engine are relevant. Extensions to our engine can deal with concurrently occurring faults by using the monitor coverage graphs to automatically construct a model with k *independent* faults per fault hypothesis. The number of such automatically generated composite fault hypotheses grows exponentially with k. However, our recovery algorithms enable rapid system recovery, and in practice, a small value of k is sufficient.

Systems Extensions. In addition to the monitors and restart actions we have considered, many other types of monitoring and recovery mechanisms can also be integrated into our framework. Models for these mechanisms will require significant expertise to construct. An interesting direction for research is the development of techniques to automatically construct the coverage, action, and cost models. Several monitoring techniques that use statistical models of normal behaviors of a system to identify deviations can already provide estimates of their false positive and negative rates, and thus allow automatic construction of coverage models. For example, the Magpie tool [19] models a request's normal flow and resource

consumption across multiple machines, while the Pinpoint tool ([20] and [21]) tracks request paths and distributions of message exchanges between system nodes. The authors of [22] use network traces to infer causal paths through multi-tier distributed applications. In [23] and [24], system invariants on the relations between resource usage at different measurement points in the system are generated using principal components analysis. Finally, language techniques for capturing operator domain knowledge regarding the effect of recovery actions and converting it into models usable by the framework also represents a promising avenue for further work.

IX. Conclusions

In this paper, we have focused on addressing challenges such as the lack of complete monitoring information in automating recovery of realistic distributed systems. We presented two different algorithms, SSLRecover and MSLRecover, that combine diagnosis and recovery actions into an iterative process that can recover a system even when monitoring alone is not able to pinpoint the failed components accurately. While SSLRecover is simple and computationally efficient, MSLRecover can express complex recovery actions, decide when to invoke more monitoring, perform more efficient recovery, and optimize user-specified metrics during recovery. In addition to showing the efficiency and resilience of the algorithms via simulation, we also experimentally demonstrated how the approach can be applied to perform selective restart in EJB systems such that user-visible errors are reduced by an order of magnitude over traditional approaches. These results show that our approach is practical as a low-cost, high-availability solution for distributed systems.

References

[1] Y. Huang, C. Kintala, N. Kolettis, and N. Fulton, "Software rejuvenation: Analysis, module and applications," in *Proc. FTCS*, Jun 1995, pp. 381–390.

[2] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Gang, and R. Gowda, "Reducing recovery time in a small recursively restartable system," in *Proc. DSN*, Jun 2002, pp. 605–614.

[3] D. Oppenheimer, A. Brown, J. Beck, D. Hettena, J. Kuroda, N. Treuhaft, D. Patterson, and K. Yelick, "Roc-1: Hardware support for recovery-oriented computing," *IEEE Trans. on Computers*, vol. 51, no. 2, pp. 100–107, Feb 2002.

[4] J. Case, M. Fedor, M. Schoffstall, and J. Davin, "A simple network management protocol (snmp)," IETF, Request for Comments RFC 1157, May 1990.

[5] Y.-F. Chen, H. Huang, R. Jana, T. Jim, M. Hiltunen, R. Muthumanickam, S. John, S. Jora, and B. Wei, "iMobile EE - An enterprise mobile service platform," *ACM Journal on Wireless Networks*, vol. 9, no. 4, pp. 283–297, Jul 2003.

[6] E. Cecchet, A. Chanda, S. Elnikety, J. Marguerite, and W. Zwaenepoel, "Performance comparison of middleware architectures for generating dynamic web content," in *Proc. ACM/IFIP/USENIX Int. Middleware Conf.*, 2003.

[7] M. Cukier, D. Powell, and J. Arlat, "Coverage estimation methods for stratified fault-injection," *IEEE Trans. on Computers*, vol. 48, no. 7, pp. 707–723, 1999.

[8] I. Lee and R. K. Iyer, "Diagnosing rediscovered software problems using symptoms," *IEEE Trans. on Soft. Engr.*, vol. 26, no. 2, pp. 113–127, 2000.

[9] K. R. Joshi, "Stochastic-model-driven adaptation and recovery in distributed systems," Ph.D. dissertation, University of Illinois at Urbana-Champaign, May 2007.

[10] Y. Chang, L. Lander, H.-S. Lu, and M. Wells, "Bayesian analysis for fault location in homogenous distributed systems," in *Proc. SRDS*, Oct 1993, pp. 44–53.

[11] G. Monahan, "A survey of partially observable Markov decision processes: Theory, models, and algorithms," *Management Science*, vol. 28, no. 1, pp. 1–16, 1982.

[12] S. Microsystems, "Enterprise JavaBeans technology," 2007, http://java.sun.com/products/ejb.

[13] ObjectWeb, "JOnAS: Java open application server," World Wide Web, 2006, http://www.objectweb.org/jonas.

[14] G. Candea, M. Delgado, M. Chen, and A. Fox, "Automatic failure-path inference: A generic introspection technique for Internet applications," in *Proc. Third IEEE Workshop on Internet Applications (WIAPP 03)*, p. 132, 2003.

[15] F. Preparata, G. Metze, and R. Chien, "On the connection assignment problem of diagnosable systems," *IEEE Trans. on Electronic Comp.*, vol. EC-16, no. 6, pp. 848–854, Dec 1967.

[16] M. Blount, "Probabilistic treatment of diagnosis in digital systems," in *Proc. FTCS*, 1977, pp. 72–77.

[17] D. Blough and A. Pelc, "Diagnosis and repair in multiprocessor systems," *IEEE Trans. on Comp.*, vol. 42, no. 2, pp. 205–217, Feb 1993.

[18] G. Khanna, M. Y. Cheng, P. Varadharajan, S. Bagchi, M. P. Correia, and P. J. Verissimo, "Automated rule-based diagnosis through a distributed monitor system," *IEEE Trans. on Dependable and Secure Comp.*, vol. 4, no. 4, pp. 266–279, 2007.

[19] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using magpie for request extraction and workload modeling," in *Proc. OSDI*, Dec 2004, pp. 259–272.

[20] M. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem determination in large, dynamic internet services," in *Proc. DSN*, 2002, pp. 595–604.

[21] E. Kiciman, "Using statistical monitoring to detect failures in internet services," Ph.D. dissertation, Stanford University, Sep. 2005.

[22] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes." in *Proc. SOSP*, 2003, pp. 74–89.

[23] G. Jiang, H. Chen, and K. Yoshihira, "Discovering likely invariants of distributed transaction systems for autonomic system management," *Cluster Computing*, vol. 9, no. 4, pp. 385–399, 2006.

[24] H. Chen, G. Jiang, and K. Yoshihira, "Failure detection in large-scale internet services by principal subspace mapping," *IEEE Trans. on Knowledge and Data Engr.*, vol. 19, no. 10, pp. 1308–1320, Oct. 2007.

[25] A. Daidone, F. Di Giandomenico, A. Bondavalli, and S. Chiaradonna, "Hidden Markov models as a support for diagnosis: Formalization of the problem and synthesis of the solution," in *Proc. SRDS*, Oct 2006, pp. 245–256.

[26] C. Basile, M. Gupta, Z. Kalbarczyk, and R. Iyer, "An approach for detecting and distinguishing errors versus attacks in sensor networks," in *Proc. DSN*, 2006, pp. 473–484.

[27] I. Rish, M. Brodie, S. Ma, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez, "Adaptive diagnosis in distributed systems," *IEEE Trans.on Neural Networks*, vol. 16, no. 5, pp. 1088–1109, 2005.

[28] P. Bahl, R. Chandra, A. Greenberg, S. Kandula, D. Maltz, and M. Zhang, "Towards highly reliable enterprise network services via inference of multi-level dependencies," in *Proc. SIGCOMM*, Aug 2007.

[29] S. Ruan, F. Tu, and K. Pattipati, "On multi-mode test sequencing problem," in *Proc. IEEE Sys. Readiness Tech. Conf.*, Sep 2003, pp. 194–201.

[30] M. Castro and B. Liskov, "Practical byzantine fault tolerance and proactive recovery," *ACM Trans. Comp. Sys.*, vol. 20, no. 4, pp. 398–461, 2002.

[31] L. Zhou, F. B. Schneider, and R. V. Renesse, "Coca: A secure distributed online certification authority," *ACM Trans. Comput. Sys.*, vol. 20, no. 4, pp. 329–368, 2002.

[32] M. A. Marsh and F. B. Schneider, "Codex: A robust and secure secret distribution system," *IEEE Trans. on Dependable and Secure Comp.*, vol. 1, no. 1, pp. 34–47, 2004.

[33] P. Sousa, A. N. Bessani, M. Correia, N. F. Neves, and P. Verissimo, "Resilient intrusion tolerance through proactive and reactive recovery," *Proc. IEEE Pacific Rim Int. Symp. on Dependable Computing*, pp. 373–380, 2007.

[34] F. Qin, J. Tucek, J. Sundaresan, and Y. Zhou, "Rx: treating bugs as allergies—a safe method to survive software failures," in *Proc. SOSP*, 2005, pp. 235–248.

[35] H. de Meer and K. S. Trivedi, "Guarded repair of dependable sys." *Theoretical Comp. Sci.*, vol. 128, pp. 179–210, 1994.

[36] K. G. Shin, C. M. Krishna, and Y.-H. Lee, "Optimal dynamic control of resources in a distributed system," *IEEE Trans. on Soft. Engr.*, vol. 15, no. 10, pp. 1188–1198, Oct 1989.

[37] M. Littman, N. Ravi, E. Fenson, and R. Howard, "An instance-based state representation for network repair," in *Proc. 19th National Conf. on Artificial Intelligence (AAAI 2004)*, Jul 2004, pp. 287–292.

[38] S. Montani and C. Anglano, "Achieving self-healing in service delivery software systems by means of case-based reasoning," *J. Applied Intelligence*, vol. 28, no. 2, pp. 139–152, 2008.