Automatic Model-Driven Recovery in Distributed Systems

Kaustubh R. Joshi[‡] Matti A. Hiltunen[§] William H. Sanders[‡] Richard D. Schlichting[§]

[‡]Coordinated Science Laboratory
University of Illinois at Urbana-Champaign
Urbana, IL, USA
{joshi1,whs}@crhc.uiuc.edu

[§]AT&T Labs Research
180 Park Ave.
Florham Park, NJ, USA
{hiltunen,rick}@research.att.com

Abstract

Automatic system monitoring and recovery has the potential to provide a low-cost solution for high availability. However, automating recovery is difficult in practice because of the challenge of accurate fault diagnosis in the presence of low coverage, poor localization ability, and false positives that are inherent in many widely used monitoring techniques. 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. In this paper, we present two recovery algorithms with complementary properties and trade-offs, and validate our algorithms (through simulation) by fault injection on a realistic e-commerce system.

1 Introduction

Building computer systems that are highly available has always been critical in certain application areas, but the increased use of computer systems in new application areas (e.g., entertainment and e-commerce) requires low-cost solutions for high availability. Traditional approaches for high availability are based on the combination of redundancy and 24/7 operations support in which human operators can detect and repair failures and restore redundancy before the service provided by the system is compromised. However, both redundancy and 24/7 operations support are expensive, and this cost may be prohibitive for many application domains. Therefore, automated recovery of failed hardware and software components (especially through restart) has been gaining attention since the mid 1990s (e.g., [9]). Activity in this area has recently increased thanks to the IBM autonomic computing initiative [10], recent work on recursive restartability [4], and recovery-oriented computing [13].

However, automation of system recovery in realistic distributed systems can be complicated, and this paper addresses many of the complications. Before recovery actions can be chosen, the system must be able to determine which components have failed, and that process is fraught with difficulty. In practical systems, monitoring is often in the form of a collection of different monitoring mechanisms. For example, while some COTS components (e.g., databases or load balancers) often provide interfaces for standards-based (e.g., SNMP [5]) monitoring, other components may have no monitoring or some proprietary methods. Additionally, the overall system usually has some monitoring method to verify that the system is able to provide its service (end-to-end monitoring). However, monitoring mechanisms often have incomplete coverage and diagnosability, and are geared to alert the operators, who have to use their domain knowledge and other tools to determine what corrective action to take (e.g., which component to restart). Monitoring mechanisms may also provide conflicting information and suffer from the typical problems of false positives (e.g., due to timeout-based testing) and false negatives (e.g., a general monitoring test does not detect an application-specific problem). 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 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 diagnosable using system diagnosis [14] 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 whether 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 illustrate its use on a small, but realistic deployment of the Enterprise Messaging Network (EMN) platform developed at AT&T [7]. Finally, we provide fault injection results that compare the efficiency of our approach with a theoretical optimum.

2 Overview

In this section, we illustrate 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.

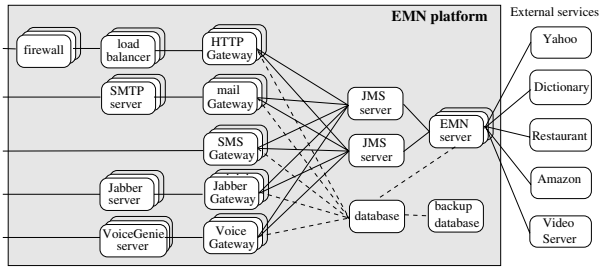

Figure 1. EMN Architecture

Motivating Example. EMN is a platform for providing services to a wide range of mobile devices such as cell phones, pagers, and PDAs via multiple protocols such as WAP, e-mail, voice, or SMS. Its architecture is shown in Figure 1 and is representative of most modern three-tiered e-commerce systems. 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, but it is currently achieved through costly 24/7 human operations support. Rapid automated fault recovery would provide both

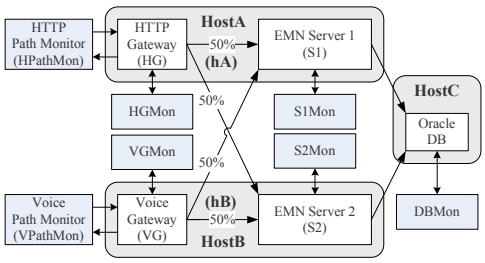

Figure 2. Simple EMN Configuration

cost and availability benefits, but several challenges need to be solved to provide it.

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 component and end-to-end path monitors. Path 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 path 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 component 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 components or hosts, starting of additional copies of certain components (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.

Running Example: The CoOL System. To highlight the key challenges and solutions for automatic recovery without overwhelming the reader with details, we illustrate our approach on a simplified (though realistic) example configuration of EMN shown in Figure 2. The configuration implements a Company Object Lookup (CoOL) system for a sales organization. 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 for-

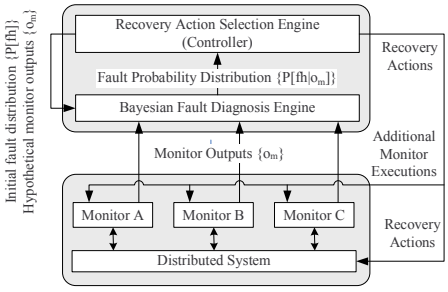

Figure 3. Automatic Recovery Architecture

ward 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 path monitors that test the overall functionality of the system via the two protocol gateways. Finally, because they are very amenable to automation, the only two classes of recovery actions we consider in this paper are restarting of components and the (more expensive) rebooting of hosts.

From the perspective of wider applicability, the CoOL system is architecturally similar to most three-tier e-commerce/web services that consist of a web server, an application server, and a back-end database. Additionally, because of their simplicity, the component and path monitors are two of the most commonly used monitoring techniques in real e-commerce applications. Designing path monitors to provide good coverage without being intrusive is an important problem, but is orthogonal to our work. Our approach can use existing path 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 and transient failures that form a large fraction of operational problems 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 an increasingly large class of practical e-commerce and business processing systems (e.g. web-services) that require high availability, but are cost-sensitive.

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 the minimum cost. This is done via a "recovery controller" whose runtime architecture is presented in Figure 3. 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 injecting 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 in the system with regard to each fault hypothesis fh, i.e., the probability that monitor m will report a failure if fh is true (Section 3). Finally, we specify the effects of each recovery action according to how it modifies the system state and fault hypothesis (e.g., restart converts a "temporary fault" hypothesis into a "no fault").

During runtime, when a failure is detected by 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 to combine monitor outputs and determine the likelihood that each fault hypotheses is true (Section 4). To do so, it uses the coverage models, historical component behavior, and the outputs of the monitors. It then invokes a recovery algorithm to choose appropriate recovery action(s). We present two different recovery algorithms: SSLRecover and MSLRecover. SSLRecover (Single Step Lookahead) is a computationally efficient greedy procedure that chooses the minimum cost recovery action that can restore the most likely fault hypothesis (Section 5). MSLRecover (Multi-Step Lookahead) is a more sophisticated algorithm 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. The algorithm is based on partially observable Markov decision processes (POMDPs) (Section 6).

3 System Model and Assumptions

During its operation, the recovery controller maintains (probabilistic) information about the presence of faults in the system through a set of "fault hypotheses" $\mathcal{FH}$. Precisely, a *fault hypothesis* $fh \in \mathcal{FH}$ is a Boolean expression that, when true, indicates the presence of one or more faults inside the system. For example, in the CoOL system, the single fault hypothesis Crash(HG) could represent a situation in which the HTTP Gateway component has crashed but all the other hosts and components are operational[1], while the composite fault hypothesis Crash(HG, DB) could indicate that both the HTTP Gateway and the database have crashed. Often, hidden inter-component dependencies and fault propagation cause cascading failures in distributed systems. We assume that the controller must deal with such

[1]The term *fault hypothesis* includes individual component failures because they are faults from the whole-system point of view.

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. In addition to user-specified fault hypotheses, the recovery controller maintains a special null fault hypothesis, fh_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 in practice, that is not a limitation, because each fault hypothesis can represent multiple faults in the system.

A set of monitors $\mathcal{M}$ provides the only way for the recovery controller to test the validity of the various fault hypotheses. Each monitor returns true if it detects a fault, and false otherwise. We assume, given that a certain fault hypothesis is true, that each monitor's output is independent both of other monitors' output and of other outputs of its own if it is invoked multiple times. 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* $\text{P}[m|fh]$, i.e., the probability that monitor $m \in \mathcal{M}$ will return true if the fault hypothesis $fh \in \mathcal{FH}$ is true. Note that false positives for monitor m can be specified simply as the probability that the monitor reports true when the null fault hypothesis fh_ϕ is true, i.e., $\text{P}[m|\text{fh}_\phi]$.

Finally, the application-specific recovery actions $a \in \mathcal{A}$ are characterized by their mean execution time (or cost), the conditions in which they are applicable, and the effect they have on the system. Multiple recovery actions may be applicable at the same time, and the control algorithm picks the best one. Because the descriptions of actions differ with the specific control algorithm used, we defer their formal description.

The recovery controller treats the fault hypotheses as opaque, and does not associate any semantics with them apart from their interaction with monitors and recovery actions. Therefore, one can always define fault hypotheses based on the available monitoring and recovery actions (e.g., "fail-silent failure in component *c* that can be fixed by restarting"). Doing so can ensure that recovery actions will always succeed without reducing the power of the recovery controller. The reason is that the controller is guaranteed to detect (and alert an operator about) faults inconsistent with the monitor coverage specifications or not fixable by the defined recovery actions whether a fault hypothesis is defined for the faults or not.

CoOL Example. For the CoOL system, we consider one fault mode for hosts, Down, and two fault modes for components, Crash and Zombie. Down(*h*) implies that host *h* has crashed (and does not respond to monitoring tests), but rebooting the host will fix the problem. Crash(*c*) means that component *c* has crashed, while Zombie(*c*) means that component *c* is alive, but cannot otherwise perform its intended function. Notice that Zombie faults are not fail-silent, and therefore cannot be detected by ping-based component monitors. However, path-based monitors work by comparing the result obtained in response to a test request with a golden value and can thus detect such non-fail-silent behaviors.

Monitor	d(hA)	c(HG)	z(HG)	c(S1)	z(S1)	z(DB)
HPathMon	1	1	1	0.5	0.5	1
VPathMon	0.5	0	0	0.5	0.5	1
HGMon	1	1	0	0	0	0
VGMon	0	0	0	0	0	0
S1Mon	1	0	0	1	0	0
S2Mon	0	0	0	0	0	0
DBMon	0	0	0	0	0	0

Table 1. Partial Monitor Coverage

The system has five ping-based component monitors (HGMon, VGMon, S1Mon, S2Mon, and DBMon) and two end-to-end path monitors (HPathMon and VPathMon) through the HTTP and Voice gateways respectively. Table 1 shows the monitor coverage for a subset of the fault hypotheses. Down(*h*) is abbreviated as d(*h*), Crash(*c*) as c(*c*), and Zombie(*c*) as z(*c*). 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 reality, timeouts that are too short also affect the coverage of component monitors (by making it less than 1) and introduce false positives, but we ignore such details with the comment that they can be easily incorporated by using network measurements in coverage estimation. In order to compute the coverage for the path monitors automatically, we use the Markov chain induced by the request flow-graph. Round-robin load balancing is represented by probabilistic transitions in the Markov chain. Finally, we observe that because of the load balancing, none of the monitors can differentiate between zombie faults in the two EMN servers, thus thwarting a "diagnose-only" approach.

Finally, two types of recovery actions are used in the CoOL system. *c*.restart() actions allow recovery from c(*c*) and z(*c*) faults by restarting of component *c*, while *h*.reboot() actions allow recovery from d(*h*) faults by rebooting of host *h*. For the CoOL example, we assume that all three types of fault hypotheses considered are temporary, and thus can always be fixed by restarting/rebooting. Although we do not do so for the CoOL example, recovery actions such as migration (i.e., restart a component on another host) can be easily added to the system (and modeled) to mask permanent faults.

4 Fault Hypothesis Estimation

When one or more monitors report a fault, the controller combines the outputs of *all* monitors into a single probabil-

ity distribution on the set of fault hypotheses using Bayesian estimation. The idea of using Bayesian estimation to deal with incomplete monitor information is not new. However, past work on fault diagnosis (e.g., [6]) has focused mainly on tests of the individual components in a system (what we refer to as *component monitors*). We believe that our uniform treatment of multiple types of realistic system monitors (e.g., path monitors) with differing characteristics, together with the coupling of diagnosis and recovery in a practically important setting, differentiates our work from previous diagnosis efforts.

In the following discussion, let o_m denote the current output of monitor $m \in \mathcal{M}$, and $o_{\mathcal{M}}$ be the current set of all monitor outputs. Let P$[fh]$ be the a priori probability that fault hypothesis fh is true. If the controller has no prior knowledge about the frequency of faults, it can assume they are equally likely and set P$[fh] \leftarrow 1/|\mathcal{FH}|, \forall fh \in \mathcal{FH}$. The probabilities can then be updated based on the outputs of the monitors and the monitor coverage specifications P$[o_{\mathcal{M}}|fh]$ using the Bayes rule as follows.

$$\textsc{Bayes}(\mathrm{P}[fh], o_{\mathcal{M}}) = \frac{\mathrm{P}[o_{\mathcal{M}}|fh]\mathrm{P}[fh]}{\sum_{fh' \in \mathcal{FH}} \mathrm{P}[o_{\mathcal{M}}|fh']\mathrm{P}[fh']} \quad (1)$$

$$\mathrm{P}[o_{\mathcal{M}}|fh] = \prod_{m \in \mathcal{M}} \mathbf{1}[o_m]\mathrm{P}[m|fh] + \mathbf{1}[\neg o_m](1 - \mathrm{P}[m|fh]) \quad (2)$$

Here, $\mathbf{1}$[expr] is the indicator function, and is 1 if expr is true, and 0 otherwise. Note that equation 2 is a result of the monitor independence assumption. Note also that if the original hypothesis probability P$[fh]$ is 0, then so is the updated hypothesis probability. We call this property the *zero preservation* property. Hence, if a fault hypothesis is known to be false (either using external knowledge, or because 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.

CoOL Example. Assume that the HTTP Gateway goes into a zombie state. Hence, HPathMon reports true, but VPathMon and all the component monitors report false. After the first Bayesian update, all d(h) and c(c) fault hypotheses are eliminated because of the false outputs of all the component monitors, and the z(DB) and z(VG) hypotheses are eliminated because the output of the VPathMon is false. Therefore, the only remaining possibilities are z(HG), z(S1), or z(S2). Because P$[o_{\mathcal{M}}|$z(HG)] = 1 and P$[o_{\mathcal{M}}|$z(S1)] = P$[o_{\mathcal{M}}|$z(S2)] = 0.25, therefore P[z(HG)] = 0.6667 and P[z(S1)] = P[z(S2)] = 0.1667 after the Bayesian update. Note that a similar result could have occurred if either z(S1) or z(S2) were true. However, another round of testing (resulting again in HPathMon returning true and VPathMon returning false) would cause the Bayesian update to update the probabilities to P[z(HG)] = 0.8889 and P[z(S1)] = P[z(S2)] = 0.0557, lending credibility to the z(HG) hypothesis. On the other hand, if z(S1) or z(S2) was true, HPathMon would return false at least once, and z(HG) would be eliminated, resulting in P[z(S1)] = P[z(S2)] = 0.5.

Improved Initial Estimates for P[fh]. In cases where all monitor coverage values (and thus the right-hand side of Equation 2) are 0 or 1, the Bayesian update in Equation 1 becomes "equality-preserving", i.e., if two hypotheses have equal initial probabilities, they have equal updated probabilities. Equality preservation can make it difficult to choose between competing failure hypotheses. However, in many cases, it is possible to overcome the problem if different fault hypotheses occur at different rates. In these cases, the recovery controller can keep track of fault hypotheses' occurrence rates, and use them to compute initial estimates for $\{\mathrm{P}[fh]\}$. For example, if fault hypotheses are assumed to occur according to a Poisson arrival process (with rate λ_{fh} for hypothesis fh), then $\mathrm{P}[fh] = \lambda_{fh} / \sum_{fh' \in \mathcal{FH}} \lambda_{fh'}$. The controller can update the values of λ_{fh} at runtime by noting when a fault occurred (i.e., the time the controller was invoked) and the action a that caused the system to recover, and updating the rates of the fault hypotheses from which action a recovers.

5 Single-Step Lookahead Recovery

Once the probability for each fault hypothesis has been computed by the Bayesian update, recovery can be attempted. In this section, we present a simple recovery algorithm, SSLRecover, that chooses the least expensive recovery action that can recover from the most likely fault hypothesis. The algorithm only looks one recovery action into the future while making its decision. SSLRecover can be used when individual recovery actions are complete in themselves, that is, each recovery action can recover from some fault hypothesis without the help of other recovery actions, and the order in which recovery actions are applied does not affect their outcomes. For the purposes of the algorithm, each recovery action $a \in \mathcal{A}$ is specified using two parameters: $a.\mathrm{FaultSet} \in 2^{\mathcal{FH}}$, which is the set of fault hypotheses from which the action can recover, and $a.\mathrm{cost} \in \mathbb{R}^+$, which is the cost of executing that action (e.g., down time induced by the action). For example, an h.reboot() action can recover from d(h), and from z(c) and c(c) of all components c residing on h.

The single-step recovery process is shown in Algorithm 4 and is invoked with an initial fault hypothesis distribution when a monitor detects a problem. The algorithm then executes the monitors and updates the probabilities of the hypotheses using Bayesian update (Equation 1). It repeats the process *maxtests* times, or until one of the following conditions occurs. First, if the probabilities of all the fault hypotheses approach zero, it means that a fault has occurred that is detected by the monitors, but is not included in the fault hypothesis set (Theorem 5.1). Second, if the probabil-

SSLRECOVER($\{P[fh]\}$,$\mathcal{FH}$,$\mathcal{M}$,$\mathcal{A}$,ϵ,maxtests)

while (true) **do**
  for i = 1 **to** maxtests **do**
    $o_{\mathcal{M}} \leftarrow$ EXECUTEMONITORS($\mathcal{M}$);
    foreach $fh \in \mathcal{FH}$ **do**
      $P[fh]_{old} \leftarrow P[fh]$; $P[fh] \leftarrow$ BAYES($P[fh], o_{\mathcal{M}}$);
    if $\nexists fh \in \mathcal{FH}$, s.t. $P[fh] > \epsilon$ **then return alert**;
    if $P[fh_\phi] \geq 1 - \epsilon$ **then return success**;
    if $\max_{fh \in \mathcal{FH}}(P[fh] - P[fh]_{old}) < \epsilon$ **then break**;
  end for
  MaxFh $\leftarrow \{\arg\max_{fh \in (\mathcal{FH} - fh_\phi)}(P[fh])\}$;
  PosActions $\leftarrow \{a \in \mathcal{A}$ s.t. $(\text{MaxFh} \cap a.\text{FaultSet} \neq \phi)\}$;
  Action $\leftarrow \arg\min_{a \in \text{PosActions}}(a.\text{cost})$;
  EXECUTEACTION(Action);
  foreach $fh \in$ Action.FaultSet **do**
    $P[fh_\phi] \leftarrow P[fh_\phi] + P[fh]$; $P[fh] \leftarrow 0$;
end while

Figure 4. Single-Step Lookahead Recovery

ity 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). Third, if the fault hypotheses probabilities do not change by more than a specified tolerance value (ϵ), it is unnecessary to repeat the iteration again, and the algorithm can move to choosing the recovery action. The repeated invocation of the monitors and the Bayesian estimation is necessary because of the probabilistic nature of some monitors; for example, the HPathMon monitor in the CoOL example detects the failure of a single EMN server only half the time. The value of *maxtests* needed to detect a fault hypothesis accurately the first time is 1 in many cases (e.g., completely deterministic monitors), but can increase if a fault hypothesis has very low monitor coverage. However, a bad choice of *maxtests* will not endanger the termination properties of the algorithm (Theorem 5.1).

After the monitor and Bayesian update iterations, the algorithm selects the set of most likely hypotheses and chooses the lowest-cost recovery action that can recover from at least one of the most likely hypotheses. Once the chosen recovery action has been carried out, the probabilities of all the fault hypotheses in Action.FaultSet are set to zero, and the probability of a null fault hypothesis is correspondingly increased. The entire process is repeated 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. The algorithm is guaranteed to terminate in a finite number of steps either successfully or with an alert to the operator (Theorem 5.1). If the algorithm quits with an operator alert, it means either that the fault hypotheses were insufficient to capture the fault that has actually occurred, or that some recovery actions were unsuccessful in recovering from the fault.

CoOL Example. Continuing our example scenario, assume that z(S1) has occurred, and that HPathMon has returned true, but all the other monitors have returned false. If *maxtests* = 1, the Bayesian update results in P[z(HG)] = 0.6667, and P[z(S1)] = P[z(S2)] = 0.1667. Therefore, MaxFh will include only z(HG), and HG.restart() will be chosen as the cheapest action that would recover from P[z(HG)]. Consequently, the hypotheses' probabilities would be updated to $P[fh_\phi]$ = 0.6667, and P[z(S1)] = P[z(S2)] = 0.1667. However, a future round of monitor execution and Bayesian update might result in VPathMon returning true and HPathMon returning false, which would make fh_ϕ invalid and cause P[z(S1)] = P[z(S2)] = 0.5. In the worst case, if S2.restart() was chosen as the next action, the fault hypotheses' probabilities would become $P[fh_\phi]$ = 0.5, and P[z(S1)] = 0.5 after action execution. Another round of monitoring and application of the Bayesian update might cause HPathMon to return true and VPathMon to return false, thus invalidating fh_ϕ and causing $P[fh_\phi]$ = 0, and P[z(S1)] = 1. This would result in S1.restart() being chosen, and subsequently allow SSLRecover to terminate successfully. The diagnosis of z(S1) or z(S2) represents the worst possible scenario for the controller, because the provided path monitors are not capable of differentiating between the two. Nevertheless, the algorithm eventually chooses the correct action and terminates successfully.

In general, we can make the following statement regarding the algorithm SSLRecover.

Theorem 5.1 *If there exists at least one recovery action for every fault hypothesis, then algorithm SSLRecover always terminates in a finite number of steps either successfully, or with an operator alert. Furthermore, if the fault that has occurred is modeled correctly in the set of fault hypotheses $\mathcal{FH}$ and monitor definitions $\mathcal{M}$, the algorithm exits successfully. Otherwise, the algorithm terminates with an operator alert.*

Proof For proving termination in a finite number of steps, observe that according to the zero-preserving property of the Bayesian update rule (Section 4), once $P[fh] = 0$ for some fh, then it can never become non-zero again via a subsequent Bayesian update step. If $P[fh] = 0, \forall fh \in \mathcal{FH}$, then the algorithm exists via an operator alert. If fh_ϕ is the only non-zero element, then its $P[fh_\phi]$ = 1, and the algorithm exits successfully. Otherwise, there exists at least one non-zero maximally probable fault hypotheses fh. Since there exists (by assumption) at least one action that can recover from fault fh, there exists a lowest-cost action for fh. After that action is executed, $P[fh]$ is set to 0. Hence, in a single iteration of the outer loop, either the algorithm exits, or $P[fh] = 0$ for some fh for which $P[fh] \neq 0$. That, coupled with the fact that the number of fault hypotheses is

finite, implies that the algorithm terminates in a finite number of steps.

Briefly, to show successful termination of SSLRecover when a correctly modeled fault occurs, it is enough to observe that the fault will always be chosen as part of the set MaxFh during some iteration of the outer loop unless one of the following two conditions are true: the algorithm exits *successfully* prior to choosing fh (proving the successful termination property), or P$[fh] = 0$ (reflecting an inaccurate definition of P$[m|fh]$) for some $m \in \mathcal{M}$). Both these conditions contradict the assumptions of the theorem, proving the second part by contradiction. The proof of termination with an operator alert follows trivially as a consequence of the first two parts. □

6 Multi-Step Lookahead Recovery

The SSLRecover algorithm is computationally efficient and has provable termination properties, but has some inherent limitations. First, the SSLRecover procedure uses a greedy action selection criterion. However, several cost metrics of practical importance (e.g., down time and number of recovery actions) depend on the entire *sequence* of recovery actions. In some systems, the greedy approach used by SSLRecover may not be able to minimize such metrics. Examples of such systems include those in which some actions do not recover from any faults by themselves, but enable other recovery actions or provide more information (e.g., a monitor invocation). Finally, the applicability of some actions and their effects might depend on the state of the system, something that is not modeled by the SSLRecover algorithm. To address all these shortcomings, we present an algorithm called MSLRecover that is based on a Partially Observable Markov Decision Process (POMDP) construction that evaluates sequences of future recovery actions before choosing an action. MSLRecover supports multi-step, state-dependent actions and cost metrics defined over the entire recovery process.

At the core of the MSLRecover algorithm is a state-based model of the target system, fault hypotheses, and recovery actions that is defined using the following elements.

System States. The set of system states $\mathcal{S}$ is defined using a set of state-variables $\mathcal{SV}$. Each state in $\mathcal{S}$ is a unique assignment of values to each state-variable. System states are assumed to be fully observable.

Fault Hypothesis States. Fault hypotheses are encoded in a single state variable whose value reflects the fault hypothesis currently present in the system. However, because this state variable is not fully observable, its value is represented by a probability distribution over the fault hypotheses $\{\text{P}[fh]\}$

Model-State. The model-state is then a tuple (s, fh) that includes both system and fault hypothesis state. However, due to the partial observability, it is represented probabilistically as $(s, \{\text{P}[fh]\})$.

Recovery Actions. Each actions in the action set $\mathcal{A}$ is represented by a precondition, a state effect, and a fault hypothesis effect. The precondition $a.\text{pre}(s)$ is a Boolean-valued predicate that returns true for those system states in which a is allowed and false otherwise. The state effect $a.\text{sEffect}(s) \rightarrow \mathcal{S}$ specifies how a modifies the system state, while the fault hypothesis effect $a.\text{fhEffect}(s, fh) \rightarrow \mathcal{FH}$ specifies how a transforms the fault hypothesis state (e.g., transforms some fh into the null fault hypothesis).

Cost Metric. 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 $\mathbf{c}_r(s, a, fh)$ and an impulse cost $\mathbf{c}_i(s, a, fh)$ that is accrued instantaneously when the controller chooses action a in model-state (s, fh). With these definitions, the single-step cost is computed as $\hat{\mathbf{c}}(s, a, fh) = \mathbf{c}_i(s, a, fh) + \mathbf{c}_r(s, a, fh) \cdot \{a.\bar{t}(s, fh) + \text{monitor}.\bar{t}\}$ where $a.\bar{t}(s, fh)$ 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 reward or profit.

CoOL Example. For the CoOL system, the state includes the current set of hosts $\mathcal{H}$, components $\mathcal{C}$, monitors $\mathcal{M}$, and a partition components(h) $\rightarrow 2^{\mathcal{C}}$ that specifies the set of components running on each host. Making these definitions a part of the state allows recovery actions to change them (e.g., enable/disable hosts, monitors, or components). There are two classes of actions, c.restart and h.reboot, and a third "dummy" monitor-only action *mon*. All actions are always enabled (the pre predicate always returns true), and they do not have any state effects (sEffect is the identity function). fhEffect for the *mon* action is an identity function, while for c.restart it transforms the c(c) and z(c) fault hypotheses to fh_ϕ. For the h.reboot action, fhEffect transforms d(h), c(c), z(c) $\forall c \in$ components(h) to fh_ϕ. Finally, since we wish to minimize the number of user requests denied due to failure, the rate cost (assuming an equal mix of voice and HTTP requests) is defined as $\mathbf{c_r}(s, a, fh) := \mathbf{1}[fh \in \{\text{d(hC), c(DB), z(DB)}\}] + 0.5 \cdot \mathbf{1}[fh \in \{\text{c(S1), c(S2), z(S1), z(S2), c(HG), z(HG), c(VG), z(VG)}\}] + 0.25 \cdot \mathbf{1}[fh \in \{\text{d(hA), d(hB)}\}]$. The definition specifies that the fraction of missed requests is 1 if the database or its host is down, 0.5 if either Gateway is down (due to the workload mix), 0.5 if either EMN server is down (due to the load balancing), and 0.25 if HostA or HostB is down (because multiple components are affected).

We can now cast the optimal recovery problem in terms of the model elements above. Let $(S_t, \{\text{P}[fh]\}_t)$, A_t, and fh_t denote the (random) 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 al-

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

gorithm is to choose a sequence of actions $a_1,\ldots,a_T$ such that the average one-step cost summed over the entire recovery process $\mathbb{E}\left[\sum_{t=1}^{T}\hat{\mathbf{c}}(S_t,A_t,fh_t)\right]$ is minimized. Let $V(s,\{\mathrm{P}[fh]\})$ represent this minimum cost if the system starts in model-state $(s,\{\mathrm{P}[fh]\})$; we call it the *mincost* of that model-state. If the problem is cast in the POMDP framework [12] with a total-cost criterion, it is known that the mincost for all states can be computed as the solution of the Bellman dynamic programming recursion. This recursion adapted for the system model described above is given by Equation 3. Note that the equation differs from traditional POMDPs in that some states (system states) are fully observable, but others (fault hypothesis states) are not. This separation reduces the effective state-space size and allows preconditions based on system-state without invalidating any optimality guarantees.

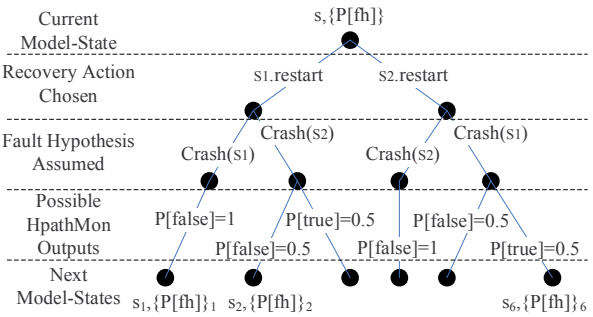

Figure 5. Example Step of the MSLRecover Computation Tree

The easiest way to understand Equation 3 is through a sample single-step of the recursion tree as shown in Figure 5. For each possible combination of fault-hypothesis fh 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 $\mathcal{O}_{fh}$ of possible monitor output combinations that could possibly be generated after a was executed with fh present in the system. For each such possible monitor output combination $o_{\mathcal{M}}$, the next system-state is just $a.\mathrm{sEffect}(s,fh)$. To compute the next fault hypothesis state corresponding to $o_{\mathcal{M}}$, we first compute how execution of action a affects the fault hypothesis probabilities, using the following equation:

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

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

$$\mathrm{P}[o_{\mathcal{M}}|s,a,fh] = \mathrm{P}[o_{\mathcal{M}}|fh' \leftarrow a.\mathbf{fhEffect}(s,fh)] \quad (5)$$

The mincost of a state if action a is executed with fault hypothesis fh 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 weighted sum over the set of next states). Averaging over $\mathcal{FH}$ then results in the mincost that results from choosing action a. The action that minimizes the mincost is the action chosen. Figure 5 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 $\mathcal{O}_{fh}$. 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 $\mathcal{O}_{fh}$. 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.

The overall operation of the MSLRecover algorithm is very similar to that of SSLRecover (Figure 4) except for a couple of important differences. Most importantly, in choosing the next action, MSLRecover uses Equation 3 and chooses the action that maximizes the right side of that equation. Second, unlike in the SSLRecover algorithm, the monitors are only invoked once before a recover action is chosen. The reasons is that monitoring actions can (and should) be added as explicit recovery actions, allowing the solution of Equation 3 to choose them explicitly only when better diagnosis is needed. Because of the similarity to the SSLRecover algorithm, the proof of termination in a finite number of steps (Theorem 5.1) also holds for MSLRecover if every action recovers from at least one fault hypothesis. However, actions allowed by MSLRecover are more general, and that condition may not always hold.

Finally, 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 $\{\mathrm{P}[fh]\}$), the recursion is noto-

riously 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 P[fh$_\phi$] $>$ 1-ϵ). That ensures that only a finite number of values of $\{\mathrm{P}[fh]\}$ 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 7): $\mathrm{V}(s, \{\mathrm{P}[fh]\}) = (1 - \mathrm{P}[\mathrm{fh}_\phi]) \cdot \max_{a \in \mathcal{A}} a.\bar{t}$. Essentially, the heuristic penalizes actions that do not move probability mass to fh$_\phi$. 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[fh]}). 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. Our future work will involve using bounds instead of heuristics to ensure strong guarantees on recovery cost. However, 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.

7 Implementation and Results

We have implemented both the recovery algorithms presented in this paper in C++ as part of a toolbox for model-driven adaptation and recovery. The SSLRecover algorithm accepts specifications of fault hypotheses, monitors, coverage probabilities, and recovery actions via a simple textual description language. However, when the MSLRecover algorithm is used, not only must the user provide a description of relevant system state, but the reward and recovery action specifications (e.g., the pre, fhEffect, 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 provided to a parser that converts them into C++ code that, when compiled together with the recovery algorithm library, provides an implementation of the recovery controller. For deployment in a system, the controller requires callbacks to be implemented to execute recovery actions and system monitors.

Although the controller algorithms could be demonstrated on a real EMN deployment, for obtaining experimental results, we chose simulation. We have implemented a simulation harness that interfaces with the real controller 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 using a Poisson arrival process with a specified rate λ_f, with each fault type being equally likely. When the system monitors detect the fault, the recovery algorithm is invoked. Monitors are invoked once every minute (i.e., faults are not always detected instantaneously), and take 5 seconds to execute. The recovery and monitoring actions generated by the recovery algorithms are simulated. However, the recovery algorithm is not simulated, but instead runs the implementation in real time.

The CoOL system described in Section 2 was used for the experiments. The only fault hypotheses that were considered were Crash and Zombie for each of the 5 components, and Down for each of the 3 hosts, for a total of 13 fault hypotheses. The system included one component monitor for each component and two end-to-end path monitors (one each for HTTP and Voice testing), for a total of 7 monitors. Finally, a Restart action for each component, a Reboot action for each host, and an explicit Test action (for the MSLRecover algorithm only) were assumed for a total of 9 actions. The Reboot actions were assumed to take a time of 5 minutes regardless of which host was being rebooted, while the restart actions took varying amounts of time between 1 minute (for the HTTP gateway) and 4 minutes (for the DB), depending on the component being restarted. Finally, 80% of the site traffic was assumed to be HTTP traffic, while 20% was assumed to be Voice traffic.

Four different configurations were considered: the SSLRecover algorithm with the number of tests 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 a "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). In reality, implementing a recovery oracle is not possible, because some fault types (e.g., Zombie faults in the CoOL system) cannot be localized perfectly in one round (or sometimes even more rounds) of monitoring. Hence, either additional monitoring is needed or incorrect recovery actions will have to be tolerated, both of which affect recovery time. Nevertheless, the oracle represents the best possible performance any algorithm could have given the types of faults and recovery actions available to the system, and serves as a good benchmark for comparison.

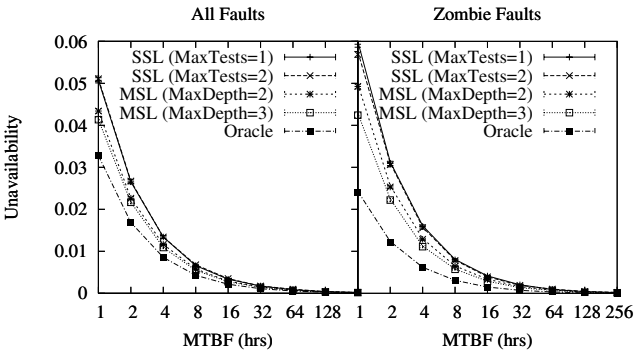

Figure 6. Unavailability under Fault Injection

The experiments were conducted on hosts with Athlon XP 2400 processors. For each simulation run, the system was run for a total time (simulated + real) of 10 days. 100 runs were performed for each configuration. Availability of the system in a simulation run was computed by integrating a fractional point measure that specified what fraction of system requests could be satisfied at each instant of time (as defined in Section 6) and dividing by the length of the interval. Therefore, multiplying the availability measure by a (constant) system request rate would return the total number of user requests satisfied. Even though the recovery algorithms support multiple (explicitly specified) faults in the system at one time, we limited fault injections to only one fault at a time for ease in determining the best oracle recovery policy (which was done by hand). Each of the five configurations was tested in two different scenarios. In one, all thirteen types of faults were injected (Crash or Zombie in each component, and Down in each host), and in the other, only Zombie faults were injected (although the recovery algorithms did not know that). Zombie faults were considered separately because they cannot be detected with good localization (since only the path monitors can detect them), and thus provide a challenge to the recovery algorithms.

The graphs in Figure 6 show the unavailability (1 - availability) of the system as a function of the MTBF when all types of faults and only Zombie component failures were injected. It can be seen that system availability remains more or less the same as the value of maxtests is increased for the SSLRecover algorithm, but increases as the value of maxdepth is increased for MSLRecover. Moreover, the system availability using the MSLRecover algorithm is better than the availability using the SSLRecover, and this difference is more pronounced for zombie-only faults. Zombie faults, unlike crash and down faults, may require multiple steps to be recovered from, and a higher lookahead has greater benefit. Finally, although the algorithms fare reasonably well with respect to the oracle benchmark, there is still room for improvement. Moreover, the difference is more pronounced for zombie faults because of their poor diagnosability.

We present more detailed measurements of the algorithms in Table 2. 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 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 2. First, the fault residual time is actually smaller than the down time because of zombie faults that have poor diagnosability. Even after a fault has been recovered from, 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 so as to bring the probability of the null fault hypothesis (fh_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 return while the fault is 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 (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 and Down faults).

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. Tech-

Algorithm (All Faults)	Faults	Cost	Down Time (sec)	Residual Time (sec)	Algorithm Time (sec)	Actions	Monitor Calls
SSLRecover (maxtests=1)	44902	199.79	290.68	226.33	0.0000803	1.837	2.89
SSLRecover (maxtests=2)	45398	199.61	291.69	227.02	0.0001146	1.802	4.653
MSLRecover (maxdepth=2)	45451	169.50	251.87	205.82	0.01318	1.590	2.868
MSLRecover (maxdepth=3)	45616	160.56	233.59	197.68	0.2210	1.428	3.079
Oracle	n/a	124.15	179.77	n/a	n/a	1.000	0.000
Algorithm (Zombie Faults)							
SSLRecover (maxtests=1)	44160	229.58	389.28	220.98	0.0001065	3.000	4.13
SSLRecover (maxtests=2)	44334	222.41	381.93	214.38	0.0000162	2.901	6.938
MSLRecover (maxdepth=2)	44726	186.44	323.40	203.64	0.03355	2.554	4.260
MSLRecover (maxdepth=3)	44939	161.94	274.13	181.00	0.4896	2.130	4.837
Oracle	n/a	84.40	132.00	n/a	n/a	1.000	0.000

Table 2. Fault Injection Results (Values are Per-fault Averages)

niques that speed up the MSLRecover algorithm and allow an increased lookahead are topics for future research.

8 Related Work

The issue of detecting failures and doing something about them has been around as long as computers. The problem was first formalized in the concept of *system diagnosis* [14]. System diagnosis involves having units test one another and using the test results to diagnose which units are faulty. The basic approach has been subsequently extended in many ways (e.g., to include probabilistic test results [3]). More closely related to our work is the extension to *sequential diagnosis* [2], within which the system is repaired incrementally through diagnosis and repair of one component at a time. The goal is to obtain a *correct sequential diagnosis* in which faulty processors are replaced but no correct processor is replaced. Despite some similarities in goals, our work is in a different vein from previous diagnosis work. We do not assume that monitors themselves can be faulty since that is usually not a problem in the kinds of systems we target. Instead, the complexity comes from having monitors of different types that have differing coverage and specificity characteristics. Furthermore, in our work, because of limited monitor coverage, it is often inevitable that correct components or hosts may be acted upon. However, we try to optimize the recovery process given the amount of information available. Finally, we allow multiple repair actions and multiple optimization criteria to drive recovery.

Closely related to our work is the theoretical work on *sequential testing* (e.g. [15]), which deals with choosing an optimal sequence of tests in order to perform diagnosis. Although that work does not deal with multiple monitor types (e.g., path monitors) that operate in parallel, our work is similar because we seek to optimize sequences of recovery actions. However, accurate diagnosis is important in our work 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. Nevertheless, due to similarities with our MSLRecover algorithm, we intend to explore whether techniques from that domain are applicable to the solution of scalability issues in our domain.

Bayesian estimation has been used in system diagnosis by other authors. For example, [6] uses Bayesian analysis in the comparison-based system analysis to deal with incomplete test coverage, unknown numbers of faulty units, and non-permanent faults. However, that work only considers one type of test (comparison between identical units), and thus does not need to address monitor coverage; it also does not consider repair.

Recently, there has been some work on performance debugging for black-box systems using passive monitoring. For instance, Aguilera et al. in [1] have developed techniques to construct causal paths through multi-tier distributed applications. Such techniques effectively provide an automated way to construct coverage models from which our path monitors could benefit, and that could work in conjunction with our approach. This is an avenue we intend to pursue in future work.

The idea of learning the effectiveness of a repair action by trying it out and then observing its effect in the system has recently been used in [11]. That work presents a learning algorithm for the construction of a *repair policy* that specifies the test and repair actions to be taken based on the outputs (true or false) of the previous test and repair actions. After the repair policy has been constructed in the learning phase, the policy remains fixed during the actual runtime of the system. By contrast, the repair decisions in our approach are made dynamically at runtime, a tactic that provides two benefits. First, because repair policies do not have to be precomputed and stored, much larger state spaces can be accommodated. Second, the system can continue to update information about failure rates and use it to make better decisions as time progresses. Finally, our work also deals specifically with non-deterministic monitor outputs, and allows repair decisions to be made based on fully observable state as well.

Automatic restarting of components, before or after they have failed, has been considered in a number of systems. [9] presents a mechanism to restart failed processes by having watchdog process `watchd` monitor a local process, and if the process dies, restart it. Recently, recovery through restart has regained attention with recursive restartability [4] and the Recovery Oriented Computing project [13]. However, most of the work on recovery through restart focuses on the mechanisms by which recovery can be automated and made more efficient, rather than determining when and where the recovery actions should be applied. In that sense, that work is complementary to our work.

Finally, there has been some work in applying techniques from Markov decision theory to dependability problems. Specifically, both [8] and [16] look at the problem of when to invoke a (human) repair process to optimize various metrics defined on the system. In both cases, the "optimal policies" that specified when the repair was to be invoked (as a function of system state) were computed off-line through solution of Markov Decision process models of the system. However, both assumed full knowledge of the state of the system, including complete knowledge of what components have failed, and thus had no notion of monitors. In addition to other differences (e.g., online solution), our approach does not share that limitation.

9 Conclusions

We have presented a holistic approach for automating the recovery of practical distributed systems even with incomplete information about what faults may have occurred in the system. We presented two different algorithms, SSLRecover and MSLRecover, that use Bayesian estimation to probabilistically diagnose faults and use the results to generate recovery actions. The algorithms are complementary in their strengths. While SSLRecover is computationally very efficient and has provable termination properties, MSLRecover can express complex recovery actions, perform more efficient recovery, and optimize user-specified metrics during recovery, but is computationally much more expensive. Finally, we compared the performance of both algorithms with a theoretically optimal (but unrealizable) Oracle via extensive fault injections, and showed that while both algorithms performed reasonably well compared to the optimum, the performance of MSLRecover was superior due to a longer lookahead. However, we believe that both algorithms are very practical, and as part of a low-cost, high-availability solution, can provide significant benefit to a large number of practical distributed systems.

Acknowledgments This work was supported in part by NSF under Grant No. CNS-0406351 and Joshi was supported by an AT&T Virtual University Research Initiative (VURI) fellowship. The authors would like to thank Jenny Applequist for improving the readability of this paper.

References

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Symposium on Operating Systems Principles*, pages 74–89, 2003.

[2] D. Blough and A. Pelc. Diagnosis and repair in multiprocessor systems. *IEEE Trans. on Comp.*, 42(2):205–217, Feb 1993.

[3] M. Blount. Probabilistic treatment of diagnosis in digital systems. In *Proc. of the 7th Symposium on Fault-Tolerant Computing*, pages 72–77, 1977.

[4] G. Candea, J. Cutler, A. Fox, R. Doshi, P. Gang, and R. Gowda. Reducing recovery time in a small recursively restartable system. In *Proc. of the 2002 International Conference on Dependable Systems and Networks*, pages 605–614, Washington, DC, Jun 2002.

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

[6] Y. Chang, L. Lander, H.-S. Lu, and M. Wells. Bayesian analysis for fault location in homogenous distributed systems. In *Proc. of the 12th Symposium on Reliable Distributed Systems*, pages 44–53, Oct 1993.

[7] 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*, 9(4):283–297, Jul 2003.

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

[9] Y. Huang and C. Kintala. A software fault tolerance platform. In B. Krishnamurthy, editor, *Practical Reusable Unix Software*, pages 223–245. John Wiley, 1995.

[10] J. Kephart and D. Chess. The vision of autonomic computing. *IEEE Comp.*, 36(1):41–50, Jan 2003.

[11] M. Littman, N. Ravi, E. Fenson, and R. Howard. An instance-based state representation for network repair. In *Proc. of the 19th National Conference on Artificial Intelligence (AAAI 2004)*, pages 287–292, Jul 2004.

[12] G. Monahan. A survey of partially observable Markov decision processes: Theory, models, and algorithms. *Management Science*, 28(1):1–16, 1982.

[13] 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*, 51(2):100–107, Feb 2002.

[14] F. Preparata, G. Metze, and R. Chien. On the connection assignment problem of diagnosable systems. *IEEE Trans. on Electronic Comp.*, EC-16(6):848–854, Dec 1967.

[15] S. Ruan, F. Tu, and K. Pattipati. On multi-mode test sequencing problem. In *Proc. of IEEE Systems Readiness Technology Conf.*, pages 194–201, Sep 2003.

[16] K. G. Shin, C. M. Krishna, and Y.-H. Lee. Optimal dynamic control of resources in a distributed system. *IEEE Trans. on Software Eng.*, 15(10):1188–1198, Oct. 1989.