How to Keep Your Head above Water While Detecting Errors

Ignacio Laguna, Fahad A. Arshad, David M. Grothe, and Saurabh Bagchi

Dependable Computing Systems Lab (DCSL)
School of Electrical and Computer Engineering, Purdue University
{ilaguna,faarshad,dgrothe,sbagchi}@purdue.edu

Abstract. Today's distributed systems need runtime error detection to catch errors arising from software bugs, hardware errors, or unexpected operating conditions. A prominent class of error detection techniques operates in a stateful manner, i.e., it keeps track of the state of the application being monitored and then matches state-based rules. Large-scale distributed applications generate a high volume of messages that can overwhelm the capacity of a stateful detection system. An existing approach to handle this is to randomly sample the messages and process a subset. However, this approach, leads to non-determinism with respect to the detection system's view of what state the application is in. This in turn leads to degradation in the quality of detection. We present an *intelligent sampling* algorithm and a *Hidden Markov Model (HMM)-based* algorithm to select the messages that the detection system processes and determine the application states such that the non-determinism is minimized. We also present a mechanism for selectively triggering computationally intensive rules based on a light-weight mechanism to determine if the rule is likely to be flagged. We demonstrate the techniques in a detection system called *Monitor* applied to a J2EE multi-tier application. We empirically evaluate the performance of Monitor under different load conditions and error scenarios and compare it to a previous system called Pinpoint.

Keywords: Stateful error detection, High throughput distributed applications, J2EE multi-tier systems, Intelligent sampling, Hidden Markov Model.

1 Introduction

1.1 Motivation

Increased deployment of high-speed computer networks has made distributed applications ubiquitous in today's connected world. Many of these distributed applications provide critical functionality with real-time requirements. These require online error detection functionality at the application level.

Error detection can be classified as *stateless* or *stateful* detection. In the former, detection is done on individual messages by matching certain characteristics of the message, for example, finding specific signatures in the payload of network packets. A more powerful approach is stateful error detection, in which the error detection

J.M. Bacon and B.F. Cooper (Eds.): Middleware 2009, LNCS 5896, pp. 205–225, 2009.

system builds up knowledge of the application state by collecting information from multiple application messages. The stateful error detection system then matches behavior-based rules, based on the application's state rather than on instantaneous information. For simplicity, we refer to stateful error detection as just *detection* in this paper.

Stateful detection is looked upon as a powerful mechanism for building dependable distributed systems [1][10]. However, scaling a stateful detection system with increasing rate of messages is a challenge. The increasing rate may happen due to a greater number of application components or increasing load from existing components. The stress on the detection system is due to the increased processing load of tracking the application state and performing rule matching. The rules can be heavy-duty and can impose large overhead for matching. Thus the stateful detection system has to be designed such that the resource usage, primarily computation and memory, is minimized. Simply throwing more hardware at the problem is not enough because applications also scale up demanding more from the detection system.

In prior work, we have presented *Monitor* [10] which provides stateful detection by observing the messages exchanged between application components. Monitor has a breaking point in terms of the rate of messages it has to process. Beyond this breaking point, there is a sharp drop in accuracy or rise in latency (i.e., the time spent in rule matching) due to an overload caused by the high incoming rate of messages. All detection systems that perform stateful detection are expected to have such a breaking point, though the rate of messages at which each system breaks will be different. For example, the stateful network intrusion detection system (NIDS) Snort running on a general-purpose CPU can process traffic up to 500 Mbps [15]. For Monitor, we have observed that the breaking point on a standard Linux box is around 100 packets/sec [10].

We have shown in previous work [11] that we can reduce the processing load of a stateful detection system by randomly sampling the incoming messages. The load per unit time in a detection system is given by the *incoming message rate* × *processing overhead per message*. Thus, processing only a subset of messages by sampling them reduces the overall load. However, sampling introduces *non-determinism* in the detection system. In sampling mode, messages are either sampled (and processed) or dropped. When a message is dropped, the detection system loses track of which state the application is in. This causes inaccuracies in selecting the rules to match because the rules are based on the application state (and the observed message). This leads to lower quality of detection, as measured by accuracy (the fraction of actual errors that is detected) and precision (the complement of false alarms).

1.2 Our Contributions

— **Intelligent Sampling:** We propose an intelligent sampling technique to reduce the non-determinism caused by sampling in stateful detection systems. This technique is based on the observation that in an application's Finite State Machine (FSM), a message type can be seen as a state transition in multiple states. If the system selectively samples and processes the messages with a high discriminating property, i.e., ones that can narrow down which state the application is in, this would limit the non-determinism.

– **Probabilistic State Determination:** Even with the proper selection of messages, there is remaining non-determinism about the application state. We propose a Hidden Markov Model (HMM)-based technique to estimate the likelihood of the different application states, given an observed sequence of messages, and perform rule matching for only the more likely states.
– **Efficient Just-in-Time Rule Matching:** We propose a technique for selectively matching computationally expensive rules. These rules are matched only when evidence of an imminent error is observed. Instability in the system, which is detected through a light-weight mechanism, is taken as evidence of such an imminent error.

We show that the three techniques make Monitor scale to an application with a high load, with only a small degradation in detection quality.

For the evaluation, we use a J2EE multi-tier application, the Duke's Bank application [12], running on Glassfish [13]. We inject errors in pairs of the combination (*component*, *method*), where 'component' can be a Java Server Page (JSP), a servlet, or an Enterprise Java Bean (EJB), and 'method' is a function call in the component. The injected errors can cause failures in the web interaction in which this combination is touched, for example, by delaying the completion of the web interaction or by prematurely terminating a web interaction without the expected response to the user. Our comparison points are Pinpoint [7] for detecting anomalies in the structure of web interactions and Monitor with random sampling [11].

The rest of the paper is organized as follows. In Section 2 we present background material on stateful detection. In Sections 3 and 4, we present the intelligent sampling and HMM-based application state estimation algorithms. In Section 5 and 6 we explain our experimental testbed, experiments and results for the intelligent sampling and HHM-based techniques. In Section 7 we present our efficient rule matching technique. In Section 8 we review related work and in Section 9 we present the conclusions, limitations of this work and future directions.

2 Background

In previous work we developed Monitor, a framework for online error detection in distributed applications [10]. Online implies the detection happens when the application is executing. Monitor observes the messages exchanged between the application components and thereby performs error detection under the principle of *black-box instrumentation*, i.e., the application does not have to be changed to allow Monitor to detect errors.

2.1 Fault Model

Monitor can detect any error that manifests itself as a deviation from the application's model and expected behavior that is given to the Monitor as input—an FSM and a set of application-level behavior-based rules. The FSM can be generated from a human-specified description (e.g., a protocol specification), or from analysis of application observations (e.g., function call traces, as done here). We define a *web interaction* as the set of inter-component messages that are caused by one user request. The end point of the interaction is marked by the response back to the user. In the context of

component-based web applications, an FSM is used to pinpoint deviations in the structure of the observed web interactions, while rules are used to determine deviations from the expected normal behavior of application's components.

2.2 Stateful Detection

Monitor architecture consists of three primary components, as shown in **Fig. 1**: the `PacketCapturer` engine, the `StateMaintainer` engine, and the `RuleMatching` engine. The `PacketCapturer` engine is in charge of capturing the messages exchanged between the application components, which can be done through middleware forwarding (as done here) or through network assist (such as, port forwarding or using a broadcast medium). When Monitor receives a rate of incoming messages close to the maximum rate that it can handle, the `PacketCapturer` is responsible for activating a sampling mechanism to reduce the workload for state transition and rule matching [11].

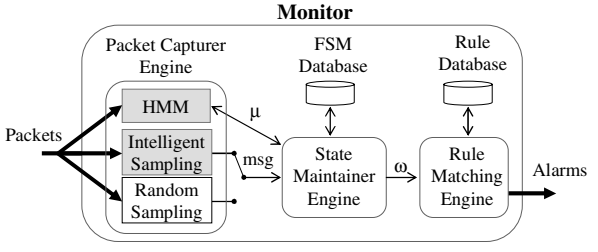

Fig. 1. Monitor architecture. One-sided and two-sided arrows show unidirectional and bidirectional flow of information respectively. Gray boxes indicate new components added to Monitor in this work.

An incoming message into Monitor may be *sampled*, meaning, it will be processed (by performing a state transition and matching rules based on that message), or it may be *dropped*. In random sampling, messages are sampled randomly without looking at the type or content of the message. As shown in [10], under non-sampling conditions, Monitor's accuracy and precision suffer when the rate of incoming messages goes above a particular point which is denoted as R_{th}. Therefore, random sampling is activated at any rate $R > R_{th}$, in which Monitor drops messages uniformly.

Sampled messages are passed to the `StateMaintainer` engine to perform state transitions according to the FSM. For each received message, the `StateMaintainer` engine is in charge of determining which states the application may be in. This is called the *state vector* and represented by ω. Here, the *events* are messages from the application that are observed at Monitor. When Monitor is in non-sampling mode, the state vector typically contains only one state ($|\omega|=1$) since Monitor has an almost-complete view of the events generated in the application—some states that do not involve externally visible messages will not be revealed to Monitor, thus ω will not always reflect the current state of the application. However, when sampling mode is activated, Monitor loses track of the actual state of the application since it is not observing every event generated by the application. Then, ω becomes a set of the possible states in which the application *can* be in. Once a message m is sampled, ω is updated. This is performed by observing (in the FSM) the new state (or states) to

where the application could have moved, from each state in ω given m. We define this mechanism as *pruning* the state vector and it is explained in further detail in Section 3.1. Typically, when ω is pruned, its size is reduced. The `RuleMatching` engine is responsible for matching rules associated with the state(s) in ω. In previous work [10] we developed a syntax for rule specification for message-based applications. We now extend the syntax to be more flexible so that it can be applied more naturally to RPC-style component-based applications. For detecting performance problems in distributed applications, we use a set of *temporal rules* that characterize allowable response time of subcomponents, i.e., the lower bound and upper bound for response time of each subcomponent. We consider that the issue of how to generate appropriate rules is outside the scope of this paper. If `RuleMatching` engine determines that the application does not satisfy a rule, we say the rule is flagged, implying the error is detected.

A challenge in Monitor, when performing random sampling, is to maintain high levels of accuracy and precision even while dropping messages. Due to the randomness of the sampling approach proposed in [11], we obtained a maximum accuracy of 0.7 when detecting failures in TRAM, a reliable multicast protocol. Systems running critical services often demand higher levels of accuracy while having low detection latency.

2.3 Building FSM from Traces

We build an FSM for the Duke's Bank Application from traces when the application is exercised with a given workload. A state S_i in the FSM is defined as a tuple (*component*, *method*). In the rest of the paper we use the term *subcomponent* to denote the tuple (*component*, *method*). This level of granularity allows Monitor to pinpoint performance problems or errors in particular methods, rather than only in components. A state change is caused by a *call* or *return* event between two subcomponents. We create the FSM by imposing a workload on the application which consists of as nearly an exhaustive list of transactions supported in the application as possible. We cannot claim this *is* exhaustive since it is manually done and no rigorous mechanism is used to guarantee completeness. When we generate application traces, no error injection is performed and we assume that design faults in the application, if any, are not activated, an assumption made in many learning-based detection systems [4][7][16]. For large-scale distributed applications, the traces may grow large, but this does not pose a significant problem because the process is offline and traces can be stored on tertiary storage and parts of them can be cached in an as-needed basis.

3 Handling High Streaming Rates: Intelligent Sampling

3.1 Sampling in Monitor

With increasing incoming message rates, Monitor opts for sampling (and dropping) messages to maintain acceptable detection latency. When a message is dropped, Monitor cannot determine the correct application state, resulting in an undesirable condition, which we call *state non-determinism*. As an example, consider an FSM

fragment in **Fig. 2**. Suppose that the application is in state S_A at time t_1, and that a message is dropped. From the FSM, Monitor determines that the application can be in state S_B or state S_C, so the state vector $\omega = \{S_B, S_C\}$. If another message is dropped at time t_2, ω grows to $\{S_B, S_D, S_E, S_F\}$.

Monitor's `RuleMatching` engine matches rules for all the states in ω. To avoid matching rules in incorrect states, Monitor prunes invalid states from the state vector once a message is sampled. For example, if the current state vector is $\{S_B, S_C\}$ and message m_2 is sampled, the state vector is reduced to $\{S_B\}$ because this is the only possible transition from any state in the state vector given the event m_2, assuming that the sampled message is not erroneous. The HMM-based algorithm (Section 4) handles the case when the sampled message may be erroneous.

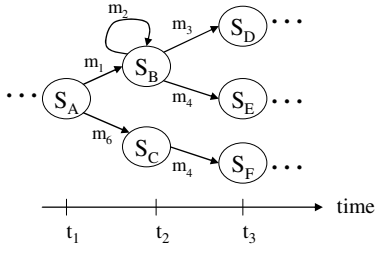

Fig. 2. A fragment of a Finite State Machine (FSM) to demonstrate non-determinism introduced by sampling

A large state vector increases the computational cost since a larger number of potentially expensive rules have to be matched leading to high detection latency. For example, an expensive rule we encounter in practice is checking consistency of multiple database tables. Worse, a large and inaccurate state vector degrades the quality of detection through an increase in false alarms and missed alarms. Our goal is then to keep the state vector size bounded so that the detection latency does not exceed a threshold (L_{th}), and the detection quality stays acceptable.

3.2 Intelligent Sampling Approach

We hypothesize that sampling based on some inherent property of messages from the FSM can lead to a reduction in the state vector size when pruning is performed. We have observed that messages in the application have different properties with respect to the different transitions in the FSM that they appear in. For example, some messages can appear in multiple transitions while others appear in only one. Suppose for example that state vector $\omega = \{S_B, S_C\}$ at time t_2 following **Fig. 2**. If m_3 is sampled, `StateMaintainer` would prune ω to $\{S_D\}$, while if m_4 is sampled, ω would be pruned to $\{S_E, S_F\}$. Thus, the fact that m_3 appears in one transition while m_4 appears in two ones, makes a difference to the resulting state vector. We say therefore that m_3 has a more desirable property than m_4 in terms of sampling.

We use an *intelligent sampling* approach whereby all incoming messages are *observed*, and a subset of messages with a desirable property is *sampled*; the others are *dropped*. A message is observed by determining its type at the application level, which determines the transition in the FSM. For our application, type is given by the combination (component, method, calllreturn). Let us define *discriminative size* d_m as the number of times a message m appears in a state transition to different states in the FSM. In the intelligent sampling approach, a message with a small d_m is more likely to be sampled. The discriminative sizes of all messages can be determined by considering the message labels on edges that are incoming into the states of the FSM.

3.3 Intelligent Sampling Algorithm

To guarantee that the rate of messages processed by Monitor is less than R_{th}, it samples n messages in a window of m messages, where $n < m$ and the fraction n/m is determined by the incoming message rate. Now, given a window of m messages, which particular messages should Monitor sample? Ideally, Monitor should wait for n messages with a discriminative size less than a particular threshold d_{th}. However, since we do not know in advance what the discriminative sizes of messages in the future will be, Monitor could end up with no sampled messages at all by the end of the window. To address this, Monitor tracks the number of messages seen in the window and the number of messages already sampled in counters *numMsgs* and *numSampled* respectively. If Monitor reaches a point where the number of remaining messages in the window ($m - numMsgs$) is equal to the number of messages that it still needs to sample ($n - numSampled$) all the remaining messages ($m - numMsgs$) are sampled without looking at their discriminative sizes. We call this point the *last resort point*. Before reaching the last resort point, Monitor samples only those messages with discriminative sizes less than d_{th}; after that, it samples all remaining messages in the window. Because of lack of space we omit the pseudocode of the intelligent sampling algorithm. The interested reader can find the pseudocode in [22].

4 Reducing Non-determinism: HMM-Based State Vector Reduction

There are two remaining problems when pruning the state vector with the intelligent sampling approach. First, when a message is sampled and the state vector is pruned, the size of the new state vector can still be large making detection costly and inaccurate. This situation arises if the FSM has a large number of states and the FSM is highly connected, or if highly discriminative messages are not seen in a window. The second disadvantage is that if the sampled message is *incorrect*, Monitor can end up with an incorrect state vector—a state vector that does not contain the actual application's state. An incorrect message is one that is valid according to the FSM, but is incorrect given the current state. For example, in **Fig. 2**, if state vector $\omega = \{S_B, S_C\}$, only messages m_2, m_3, and m_4 are correct messages. Incorrect messages can be seen due to a buggy component, e.g., a component that makes an unexpected call in an error condition. To overcome these difficulties, we propose the use of a Hidden Markov Model to determine probabilistically the current application state.

4.1 Hidden Markov Model

A Hidden Markov Model (HMM) is an extension of a Markov Model where the states in the model are not observable. In a particular state, an outcome, which is observable, is generated according to an associated probability distribution.

The main challenge of Monitor, when handling non-determinism, is to determine the correct state of the application when only a subset of messages is sampled. This phenomenon can be modeled with an HMM because the correct state of the

application is hidden from Monitor while the messages are observable. Therefore, we use an HMM to determine the probability of the application being in each of its states.

An HMM is characterized by the set of states, a set of observation symbols, the state transition probability distribution A, the observation probability distribution B (given a state i, what is the probability of observation j), and the initial state probability distribution π. We use $\lambda = (A, B, \pi)$ as a compact notation for the HMM.

We used the Baum-Welch algorithm [24] to estimate HMM parameters to model the Duke's Bank application. The HMM is trained with the same set of traces used to build the application FSM. More details about the estimation of the HMM parameters can be found in [22].

4.2 Algorithm for Reducing the State Vector Using HMM

We have implemented the `ReduceStateVector` algorithm (**Fig. 3**) for reducing the state vector using an HMM. When Monitor samples a message, it asks the HMM for the k most probable application states. Monitor then intersects the previous state vector with the set of k most probable states. Then an updated state vector is computed from the FSM using pruning (as defined in Section 2.2), i.e., by asking the FSM that given the set of states from the intersection and the sampled message, what are the possible next states.

The HMM is implemented in Monitor in the frontend thread, the `PacketCapturer`. Thus, the HMM observes all messages since they are needed to build complete sequences of observations.

The `ReduceStateVector` algorithm consists of three steps:

- **Step 1**: Calculate what is the probability that, after seeing a sequence of messages O, the application is in each of the possible states s_1, … s_N? This is expressed as $P(q_t = s_i \mid O, \lambda)$. This step produces a vector of probabilities μ_t (lines 1–3).
- **Step 2**: Sort the vector μ_t by the probability values. This produces a new vector of probabilities α_t (line 4).

`ReduceStateVector` computes a new state vector based on: the HMM, an observation sequence and a previous state vector.
Input: λ: Hidden Markov Model; O: observation sequence $O = \{O_1, O_2, \ldots, O_t\}$; ω_t: application' state vector at time t; k: Filtering criteria for the number of probabilities estimated by the HMM.
Output: ω_{t+1}
Variables: μ_t: probability vector $\mu_t = \{p_1, p_2, \ldots, p_N\}$, where $p_i = P(q_t = s_i \mid O, \lambda)$, for all i in $S = \{s_1, \ldots, s_N\}$ (the states in the FSM) and q_t is the state at time t; α_t: sorted μ_t.

```
ReduceStateVector(λ, O, ω_t, k):
1.  μ_t ← ∅
2.  For each i in S
3.      Add P(q_t = s_i | O, λ) to μ_t
4.  α_t ← sort(μ_t) by p_i
5.  I ← ∅
6.  I ← ω_t ∩ α_t[1…k]
7.  if (I = ∅) then
8.      ω_{t+1} ← ω_t ∪ α_t[1…k]
9.  else
10. ω_{t+1} ← I
11. return ω_{t+1}
```

Fig. 3. Pseudocode for reducing state vector using HMM's estimate of probability of each application state

– **Step 3**: Compute a new state vector ω_{t+1} as the intersection of the current state vector ω_t and the top k elements in α_t. By using a small k, Monitor is able to reduce the state vector to few states. If the intersection of ω_t and α_t is null, we take the union of the two sets. This is a safe choice because having the intersection of ω_t and α_t equal to null implies that either the HMM or ω_t is incorrect. We acknowledge that if both HMM and state vector are incorrect, this scheme will not work. However, proper training of the HMM makes a concurrent error highly unlikely, and one that never occurred in any of our experiments. This step is executed in lines 5-11.

Fig. 4 shows points in time when the algorithm is invoked in `StateMaintainer`. FSMLookup(ω, n) calculates the new state vector from ω given that n consecutive messages have been dropped (as explained in Section 3.1).

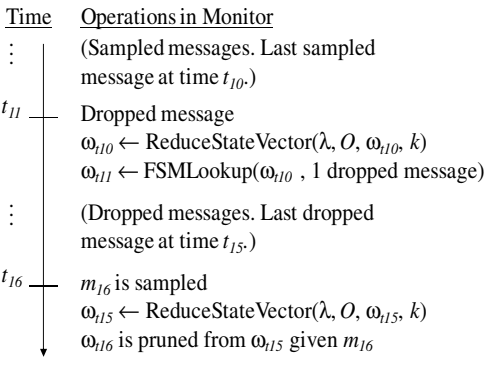

Fig. 4. Example of points in time when the ReduceStateVector algorithm is invoked

The time complexity of the algorithm is proportional to the time in computing $P(q_t = s_i | O, \lambda)$ for all the states, the time to sort the array μ_t, and the time to compute the intersection of ω_t and the top k elements in α_t. The vector μ_t can be computed in time $O(N^3T)$, where N is the number of states in the HMM (and the FSM), and T is the length of the observation sequence O. Sorting μ_t can be performed in $O(N \log N)$, and the intersection of ω_t and $\alpha_t[1\ldots k]$ can be performed in $O(Nk)$. Hence, the overall time complexity is $O(N^3T)$.

5 Experimental Testbed

5.1 J2EE Application and Web Users Emulator

We use the J2EE Duke's Bank Application [12] running on Glassfish v2 [13] as our experimental testbed. Glassfish has a package called `CallFlow` that provides a central function for Monitor—a unique ID is assigned to each web interaction. It also provides caller and called component and methods, without needing any application change.

To evaluate our solutions in diverse scenarios such as high user request rates and multiple types of workload, we developed *WebStressor*, a web interactions emulator. WebStressor takes different traces and replays them by sending each message to the tested detection systems. Each trace contains sequences of web interactions that would be seen in `CallFlow` when a user of Duke's Bank application is executing

multiple operations. WebStressor also has error injection capabilities which are explained in Section 6.3.

5.2 Pinpoint Implementation

We implemented Pinpoint [7] that proposes an approach for tracing paths through multiple components, triggered by user requests. A Probabilistic Context Free Grammar (PCFG) is used to model normal path behavior and to detect anomalies whenever a path's structure does not fit the PCFG. A PCFG has productions represented in Chomsky Normal Form (CNF) and each production is assigned a probability after a training phase. Pinpoint-PCFG is trained using the same traces from Duke's Bank that are used to build the FSM and to train the HMM. We call this implementation Pinpoint-PCFG in the paper.

6 Experiments and Results

In this section we report experiments to evaluate the performance of Monitor and compare it with that of the Pinpoint-PCFG algorithm. When we refer to Monitor, we mean baseline Monitor [10], with the addition of two techniques intelligent sampling and HMM. The machines used have 4 processors, each an Intel Xeon 3.4 GHz with 1024 MB of memory and 1024 KB of L1 cache. All experiments are run with exclusive access to the machines. We show 95% confidence intervals for some representative plots, but not all, to keep the graphs readable.

6.1 Benefits of Intelligent Sampling

We run experiments to verify our hypothesis that intelligent sampling helps in reducing the size of the state vector ω. For this, we run WebStressor with a fixed moderate user load (8 concurrent users) and with no error injection. When a message is dropped, ω increases or stays constant. When a message is sampled, ω is pruned and it is passed to the `RuleMatching` engine.

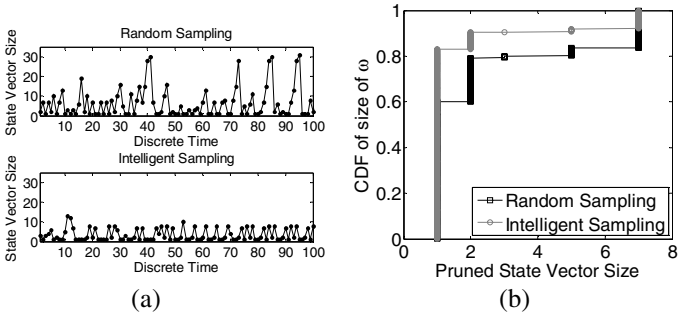

Fig. 5. Performance results when comparing Monitor random sampling and intelligent sampling. (a) Sampled values of state vector ω for Monitor with random and intelligent sampling; (b) CDF for the pruned state vector ω with random and intelligent sampling.

In each mode, we obtained 3337 sample values of ω's size. **Fig. 5**(a) shows 100 snapshots of these values for Random Sampling (RS) and Intelligent Sampling (IS) modes. Here the size of ω is shown for every message arriving at Monitor. The high-peaks pattern that we observe in RS mode is due to the deficiency of random sampling in selecting messages with small discriminative size. In contrast we do not observe this pattern in IS mode, because it preferentially samples the discriminating messages, producing smaller pruned state vectors ω.

Next, we measure ω's size only *after* it is pruned. Recall that the pruned state vector ω is the one used for rule instantiation and matching. Hence, it is at this point that it is critical to have a small ω. **Fig. 5**(b) shows the cumulative distribution function (CDF) for the observed values of ω's size. In IS mode, ω's size of 1 has a higher frequency of occurrence (about 83%) than in RS mode (60%). In contrast, all ω's size values > 1 have higher frequency of occurrence in RS than in IS. After being pruned, ω can have a maximum size of 7. This is due to the nature of Duke's Bank application in which the maximum discriminative size of a message is 7.

6.2 Definition of Performance Metrics

We introduce the metrics that we use to evaluate detection quality. Let W denote the entire set of web interactions generated in the application in one experimental run. For W, we collect the following variables, I: out of W, the web interactions where faults were injected; D: out of W, the web interactions in which Monitor detected a failure; C: out of I, the web interactions in which Monitor detected a failure (these are the correct detections).

Based on these variables, we calculate two metrics:

$$Accuracy = |C| / |I|; \; Precision = |C| / |D|$$

Accuracy expresses how well the detection system is able to identify the web interactions in which problems occurred, while precision is a measure of the inverse of false alarms in the system.

Another performance metric is the latency of detection. Let T_i denote the time when a fault is injected and T_d the time when the failure caused by the injected fault is detected by the detection system. We define *detection latency* as $T_d - T_i$. When a delay δ is injected (emulating a performance problem in a component of the application), δ is subtracted from the total time since it represents only a characteristic of the injected fault and not the quality of the detection system.

6.3 Error Injection Model

Errors are injected by WebStressor at runtime when mimicking concurrent users. This results in errors in the application traces which are fed to the detection systems. We inject four kinds of errors that occur in real operating scenarios:

1. *Response delay*: a delay d is selected randomly between 100 msec and 500 msec, and is injected in a particular subcomponent. This error simulates subcomponent's response delays due to performance problems.

2. *Null Call:* a called subcomponent is never executed. This error terminates the web interaction prematurely and the client receives a generic error report, e.g., HTTP 500 internal server error.
3. *Runtime Exception*: an undeclared exception, or a declared exception that is not masked by the application, is thrown. As in null calls, the web interaction is terminated prematurely and the client receives an error report.
4. *Incorrect Message Sequences:* an error that occurs for which there is an exception handler that invokes an error handling sequence. This sequence changes the normal structure of the web interaction. We emulate this by replacing the calls and returns in N consecutive subcomponents. The value of N is selected randomly between 1 and 5.

Of these, Pinpoint-PCFG cannot detect response delay errors. We perform comparative evaluation of Monitor with Pinpoint-PCFG for the other error types.

6.4 Detecting Performance Problems

We inject delays to simulate performance problems in the set of 5 subcomponents listed in **Table 1**. A category of errors that is difficult to detect is transient errors—those that are caused by unpredictable random events and that are difficult to reproduce and isolate. We want to test Monitor in detecting this category of errors. In order to mimic this scenario in our injection strategy, we inject delays only 20% of the time a subcomponent is touched in a web interaction.

Table 1. List of subcomponents (component, method) in which performance delays are injected

Name	Method	Type of Component
AccountControllerBean	createNamedQuery	EJB
TxControllerBean	deposit	EJB
/template/banner.jsp	JspServlet.service	servlet
/bank/accountList.faces	FacesServlet.service	servlet
/logon.jsp	JspServlet.service	servlet

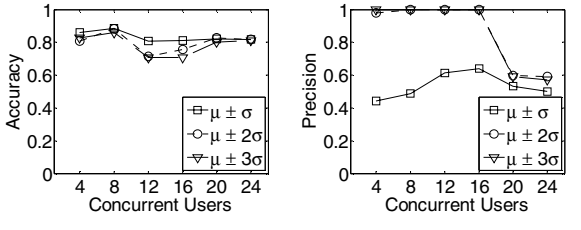

Fig. 6. Accuracy and precision of Monitor in detecting performance delays for three type of rules

Before running the experiment, we determine the best set of parameter values in Monitor. We generate ROC (Receiver Operating Characteristic) curves by varying their configuration parameters (i.e., number of rules) and the imposed load of users to the application. Then, we select the operational point as the one closest to the ideal point (0, 1); in case of a tie, we use the point with the better precision. Because of lack of space we omit the ROC curves; however, the reader can refer to [22] for these.

For the performance delay rules, first, we measure the average (μ) and standard deviation (σ) of the response time from the components in the application during the

training phase. We then create rules with the following thresholds for response times in each component: $\mu \pm \sigma$, $\mu \pm 2\sigma$ and $\mu \pm 3\sigma$.

Fig. 6 shows the results of this experiment. We observe that using $\mu \pm 2\sigma$ provides the best combination of accuracy and precision. For rule types $\mu \pm \sigma$ and $\mu \pm 2\sigma$ we observe a decrease in accuracy of about 10% as concurrent users are increased from 4 to 16, and an increase in the same order of magnitude as users are increased to 24. The reason for the increase in accuracy is due to the precision rate that decreases rapidly after 16 concurrent users. Because of the large rate of false alarms generated after this point, accuracy is increased as a trade-off.

We also evaluate the performance of random and intelligent sampling in detecting performance delays. For this experiment, we use similar definitions for accuracy and precision as in the previous experiments, but we change the granularity of detection from web interactions to individual subcomponents. Detection at the level of a subcomponent is helpful in diagnosis—finding the root cause of the problem—since it helps in pinpointing suspect subcomponents. The results are shown in **Fig. 7**(f). We observe that accuracy and precision are higher for IS for most loads (4–16 concurrent users). Although, for high loads (20 and 24 users), random and intelligent sampling exhibit almost the same (poor) performance.

6.5 Detecting Anomalous Web Interactions

We evaluate Monitor's performance in detecting anomalous web interactions by injecting null calls, runtime exceptions and incorrect message sequences. We also evaluate Pinpoint-PCFG's performance here.

Monitor detects anomalous web interactions at the `StateMaintainer`. If an event is unexpected according to the current state in Monitor's state vector, an error is flagged. This avoids the need for explicit rules for this type of detection. For the Duke's Bank application, if the correct state is S_c and the state vector after a message is sampled and pruning is completed, is ω, then we find empirically that in all cases $S_c \in \omega$. Thus, a detection happens at Monitor only if the message is incorrect, i.e., there is an actual fault. This gives a precision value of 1 for Monitor's detection of anomalous web interactions in Duke's Bank.

We empirically determine the best value of parameter k for the HMM-based state vector reduction algorithm. **Fig. 7**(a) shows Monitor running with different values of k while we inject anomalous web interactions. Parameter k=0 represents Monitor running without HMM. We observe that, with no HMM, in both low and high loads, accuracy is very low (about 0.4). Since Monitor with $k > 0$ performs better than with $k = 0$, this validates our design choice of using an HMM. In high load, two conditions cause Monitor to have a decreasing accuracy with increasing k. Monitor samples less often leading to an increase in the size of ω. With large k, few states get pruned and if the observed erroneous message is possible in any of the remaining states of ω, the error is not detected. Second, when the erroneous message may not be sampled, the HMM is particularly important. Increasing k effectively reduces the impact of the HMM, since even states with low probabilities given by the HMM are considered.

For the remaining experiments, we use k=1 as it allows Monitor to have the best accuracy in both low and high load. We determine the best configuration parameter

setting for Pinpoint-PCFG to get ROC curves under low and high loads. Pinpoint-PCFG's ROC curves can be found in [22].

Fig. 7(b)–(c) show the results for accuracy and precision of Monitor and Pinpoint-PCFG. We observe that on average, Monitor's accuracy is comparable to that of Pinpoint-PCFG. In Monitor, accuracy decreases for higher loads due to dropping more messages in a sampling widow. As the load increases, Pinpoint-PCFG maintains a high accuracy because it is not dropping messages—messages are being enqueued for eventual processing. However its latency of detection suffers significantly in high loads—it is in the order of seconds (**Fig. 7**(e)) while in Monitor it is in the order of milliseconds (**Fig. 7**(d)).

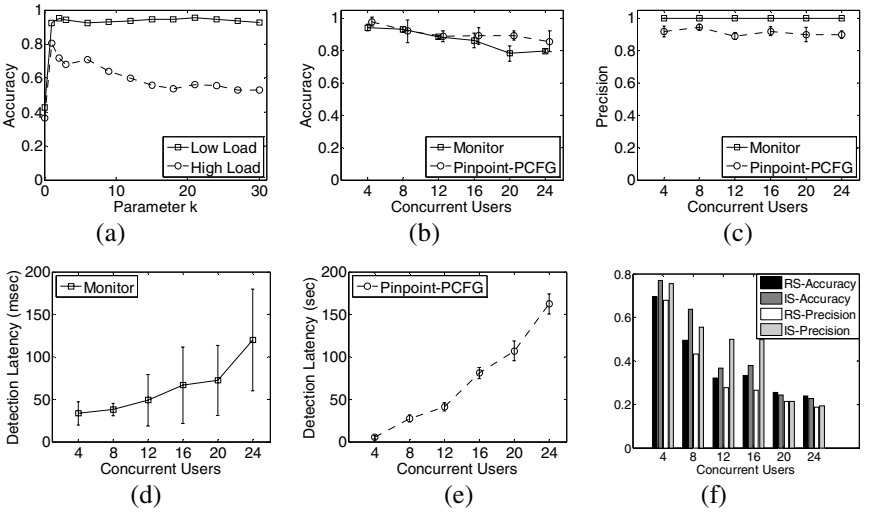

Fig. 7. Performance results for Monitor and Pinpoint when detecting anomalous web interactions. (a) Accuracy in Monitor when varying parameter *k* in the HMM-based state vector reduction algorithm; (b)–(c) Accuracy and precision for Monitor and Pinpoint-PCFG; (d)–(e)Detection latency for Monitor and Pinpoint-PCFG; (f) Accuracy and Precision for Random Sampling and Intelligent Sampling for performance delay errors.

Table 2. Memory consumption for the compared systems

	Average Memory Usage (MB)	
	Virtual Memory	Memory in RAM
Monitor	282.27	25.53
Pinpoint-PCFG	933.56	696.06

We observe the robustness of Pinpoint-PCFG to false positives as it maintains on average almost the same precision (0.9) with increasing number of users. However, the precision in Pinpoint-PCFG is lower than that in Monitor of 1.0.

The high detection latency in Pinpoint-PCFG is due to the fact that the parsing algorithm in the PCFG has time complexity $O(L^3)$ and space complexity $O(RL^2)$, where R is the number of rules in the grammar and L is the size of a web interaction. In the

Duke's Bank application we observe that the maximum length of a web interaction is 256 messages, and the weighted average size is 70. Previous work [14] has shown that the time to parse sentences of length 40 can be 120 seconds even with optimized parameters. Moreover, in Pinpoint-PCFG, error detection can only be performed after the end of web interactions which also explains longer detection latencies than in Monitor. Another cause of the high latency in Pinpoint-PCFG is the large amount of virtual memory that the process takes (933.56 MB for a load of 24 concurrent users as shown in **Table 2**). This makes the Pinpoint-PCFG process thrash.

To look into the issue of memory consumption further, we measure average memory consumption for Monitor and Pinpoint-PCFG under a load of 24 concurrent users. Physical and virtual memory usage are collected every 5 seconds by reading the `/proc` file system and averaged over the duration of each experimental run. **Table 2** shows the results of this experiment.

7 Efficient Rule Matching

7.1 Motivation

We present a technique for selectively matching computationally expensive rules in Monitor, thereby allowing it to operate under higher application message loads. The technique is based on the observation that the computationally expensive rules do not have to be matched all the time. Rather they can be matched when there is evidence of system instability. Previous work [21] has shown that errors are more likely when instability in the system is observed. For example, an increasing average response time in a web server may indicate an imminent failure because of resource exhaustion. Therefore, we use a light-weight mechanism of determining system instability to trigger the computationally expensive rules.

Many rules can be computationally expensive both in time and space. For example, a pattern matching rule such as calculating the convolution of two signals, as presented in [8], requires long computations, while matching strings with probabilistic context free grammars, as in Pinpoint [7], demands a large amount of memory space. For other rules, the system requires to re-train its model for detecting anomalies based on newly observed data, as in semi-supervised learning techniques [23]. The re-training is often quite expensive.

7.2 Selective Rule Matching Approach

We propose an approach for matching computationally expensive rules if evidence of instability is observed in the system. Instability can be observed by measuring different metrics in the application or the underlying middleware, for example, response time, memory, or CPU usage. Manifestation of instability can be in the form of abrupt changes in the measurements (either increasing or decreasing), or in fluctuations in the measurements.

Our approach for selectively matching rules is as follows. Let C_t denote the condition of the system at time t. Thus, C_t can take one of two conditions of the set {*stable*, *unstable*}; let $\{c^+, c^-\}$ denote these conditions. Suppose that at time t, a message m_t is observed, a rule R has to be matched, and a sequence of the n previously observed

messages $\{m_{t-n}, m_{t-n+1}, \ldots, m_{t-1}\}$ are kept in a buffer B. Then, if $C_t = c^-$, R is matched, otherwise, B becomes $\{m_{t-n+1}, \ldots, m_t\}$ and Monitor waits for the next message to arrive.

The main challenge in this approach is to infer and use an accurate classifier function F mapping the universe of possible messages (i.e., system-level measurements) to the range of system conditions C, so that the probability of catching an error when $C_t = c^-$ and the rule R is matched is maximized. A complete study for addressing this challenge is out of the scope of this paper and will be pursued in future work. However, we present an example in which this technique is used in Monitor for detecting a memory leak in the Apache Tomcat web server [17] by using a simple estimator of instability.

7.3 Memory Leak Injection

We instrumented the Apache Tomcat web server to inject a memory leak dynamically. Upon receiving a request, an unused object is created with probability p_{leak} in the server's thread-pool, and it is kept referenced so that it is not taken by the Java garbage collector. The result is an increase in memory usage that can be observed from the Java process running the server.

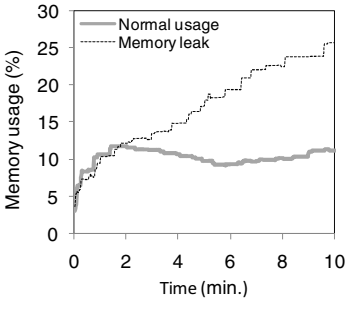

Fig. 8. Percentage of memory usage of the Apache Tomcat web server under normal conditions and with a memory leak fault injection

We perform experiments to observe the pattern of memory consumption of the web server in both normal conditions and when the memory leak is injected. We use a testbed of an e-commerce site that simulates the operation of an online store as specified by the TPC-W benchmark [18]. We use the benchmark WIPSo mixture (50% browsing and 50% ordering) that is intended to simulate a web site with a significant percentage of order requests.

Fig. 8 shows the results of the experiment when the probability p_{leak} of the memory leak injection is set to 0.5, and when a load of 50 concurrent users is imposed. Memory measurements are taken in a fixed interval of 1 second for a window of 10 minutes after the server is started.

7.4 Rule for Detecting Memory Leak Error

Previous work on software rejuvenation [19] has proposed the use of time series analysis to model memory usage patterns in the Apache web server. In this paper, we use time series analysis to build rules that are able to pinpoint a memory leak. In particular, the web server memory consumption is modeled as an autoregressive (AR) moving average (MA) process ARMA(p, q). This process is formally defined as follows [20]:

- A memory usage measurement X_t is an ARMA(p, q) process if for every time t,

$$X_t = C + \sum_{i=1}^{p} \varphi_i X_{t-i} + \varepsilon_t + \sum_{i=1}^{q} \theta_i \varepsilon_{t-i},$$

  where ε_t is the error term, C is a constant, and $\{\varphi_1, \ldots, \varphi_p\}$ and $\{\theta_1, \ldots, \theta_p\}$ are the parameters of the model.
- The error term ε_t is considered to be white noise, i.e., independently and identically distributed with mean 0 and variance σ^2.

We collect training data in several runs of the Apache Tomcat server for generating two ARMA(p, q) models λ and λ' that represent memory usage under normal conditions and memory leak conditions respectively. The models are inferred by maximum likelihood estimation by using the statistical tool R. To estimate the number of p and q parameters that best fit the models, while keeping the number of parameters small, we vary p and q over 1, 2, and 3. We then select the values of p and q that produce the minimum root-mean-square (RMS) error when comparing test data and new data generated with the models. For this, test data is labeled as being normal or erroneous when selecting the parameters in λ and λ' respectively. For our test-bed, p=3 and q=2 resulted in the best configuration for the models.

7.5 Rule Matching Latency Reduction

After the two models λ and λ' are trained, we build a rule for detecting the memory leak in the web server by observing to which model the test data fits better. The rule takes as input a sequence A of n old observed messages, and a sequence B of n new observed messages in which it will look for errors. Then, two simulated sequences S and S' are generated by using the two models λ and λ' respectively on observations A. Finally, S and S' are compared to B by measuring the RMS error. If B fits better with S', an error is flagged by the rule indicating a possible memory leak.

We detect instability in the system by measuring the standard deviation σ of the m previous observed memory consumption values and if it is greater than a threshold P_{th} we conclude $C_t = c^-$, the rule is matched.

Table 3. Detection coverage and average rule matching delay for the ARMA-based rule

Rule Matching Criteria	Memory Leak Detected	Average Matching Latency (msec.)
Always matched	yes	19.283
$\sigma \geq 0.5$	yes	7.115
$\sigma \geq 1.0$	no	1.25

Table 3 shows the results for 3 different configurations in Monitor when the memory leak is injected in the web server. For the three experiments, n=10, m = 5, and the same workload that we used for training is imposed on the web server. The initial values of 0.5 and 1.0 for σ are taken from the average standard deviation observed in the training data set for the web server running under normal conditions which is around 1.2 % of memory usage. This confirms that, in normal conditions, memory usage variation is much less than in unstable conditions.

We notice that when the rule is always matched, the average latency is the maximum as expected, and as we increase σ, the latency decreases. This is due to an inherent reduction in the chances of matching the ARMA-based rule which is more computationally expensive than evaluating σ. However, if σ is too low, the error may be missed since the ARMA-rule may not be matched at all, as is the case when σ=1.0.

Detection of the memory leak presented here can be done by many other profiling tools. The point behind this experiment is not to claim any novel detection capability. Rather, it is to show how instability can be used to trigger more computationally expensive rule matching.

8 Related Work

Error Detection in Distributed Systems: Previous approaches of error detection in distributed systems have varied from heartbeats to watchdogs. However, these designs have looked at a restricted set of errors (such as, livelocks) as compared to our work, or depended on alerts from the monitored components.

A recent work closely related to ours is Pinpoint [7]. Authors present an approach for tracing paths from user requests and use a Probabilistic Context Free Grammar (PCFG) to model normal path behavior as seen during a training phase. A path's structure is then considered anomalous if it significantly deviates from a pattern that can be derived from the PCFG. Pinpoint however does not consider the problem of dealing with high rates of requests. We provide a comparative evaluation of Monitor with Pinpoint in Section 6.5. A variant of the Pinpoint work [16] uses a weighting for long web interactions so that they are not mistakenly flagged as erroneous. This weighting seems less useful for Duke's Bank since the probabilities for the less likely transitions differ significantly from the expected probability. This work also uses an additional parameter (α) to pick a particular point in the false alarm-missed alarm spectrum. We believe that an equivalent effect is achieved through our ROC-based characterization.

Performance Modeling and Debugging in Distributed Systems: There is recent activity in providing tools for debugging problems in distributed applications, notably Project5 [8][9] and Magpie[6]. These approaches provide tools for collecting trace information at different levels of granularity which are used for automatic analysis, often offline, to determine the possible root causes of the problem.

Project5's main goal is detecting performance characteristics in black-box distributed systems. In [8] models for performance delays on RPC-style and message-based application for LAN environments are proposed—authors focus on finding causal path patterns with unexpected timing or shape. In [9] authors present an algorithm for performance debugging in wide-area systems. We determined that this work's focus is on determining the performance characteristics of different components in a complete black-box manner. Since Project 5 does not assume a uniform middleware, such as J2EE, it cannot assign a unique identifier to all messages in a causal path as they occur. We use the GlassFish-assigned unique identifier to a path of causal request-responses. In our work, we use both these features. However, Project5's accuracy suffers greatly when detecting anomalous patterns under concurrent load (in fairness,

this is not the goal of the work either). Therefore, we did not perform a quantitative comparison with Project5 for detecting performance problems (in Section 6.4).

The Magpie project [6] is complementary to our work—it is a tool that helps in understanding system behavior for the purposes of performance analysis and debugging in distributed applications. Magpie collects CPU usage and disk access for user requests as they travel though the system components. These workload models of request behavior can be used in Monitor to specify performance-based rules.

Stateful Intrusion Detection in High Throughput Streams: In the area of intrusion detection, techniques have been proposed to allow network-based intrusion detection systems (NIDS) to keep up with high network bandwidths by parallelizing the workload [1] and by efficient pattern matching [2]. Although distributing the detection load in multiple machines helps, this does not solve the fundamental problem of how to manage the resource usage in individual machines, which we address.

Sampling Techniques for Anomaly Detection: Recently there is an increased effort in finding network failures, anomalies and attacks through changes in high-speed network links. For example, in [3] authors propose a sketch-based approach, where a sketch is a set of hash tables that models data as a series of (*key*, *value*) pairs; key can be a source/destination IP address, and the value can be the number of bytes or packets. A sketch can provide accurate probabilistic estimates of the changes in values for a key. Sampling has also been used in high-speed links as input for anomaly detection [4], for example, for detecting denial-of-service (DoS) attacks or worm scans. However, some studies show that these sampling techniques introduce fundamental bias that degrades performance when detecting network anomalies [5]. Our work matches rules based on aggregated information at the application level, while this work matches rules based on network level traffic statistics of the traffic.

9 Conclusions and Limitations

This paper presents an intelligent sampling algorithm and an HMM-based technique to enable stateful error detection in high throughput streams. The techniques are applied and tested in the Monitor detection system and provide a high quality of detection (accuracy and precision) for a range of real-world errors in distributed applications with low detection latency. It compares favorably to an existing detection system for distributed component-based systems called Pinpoint. We also present a technique to optimize the cost of matching computationally expensive rules for detecting resource exhaustion. Our technique relies on triggering the expensive rules only on detecting, through lightweight means, evidence of system instability.

The techniques were tested successfully in Dukes's Bank (an online banking application) and in the Apache Tomcat web server, and they can be applied to distributed systems that are composed of multiple interacting components. In general the advantage of Monitor would be the highest when messages are discriminating in terms of state transitions to different extents in the application's FSM.

A disadvantage of our HMM-based technique is that an application with a large number of states can make the HMM processing too expensive. It is a subject of future work to determine what size of the FSM would cause a cross-over beyond which

HMM execution will have to be done with an incomplete sequence of messages, which will call for a novel algorithm itself. Another limitation of Monitor is that in sampling mode some states may not be examined. If such a state happens to contain the error condition, Monitor will miss the error. In future work we will address this problem by developing a sampling scheme that allows Monitor to preferably sample messages (or sequence of messages) that are likely to point to errors in the application. We will also work on automatic generation of rules from traces that can be obtained in previous runs of the applications, and on scaling the matching of different computationally expensive rules.

Acknowledgements

The authors would like to thank Patrick Reynolds for discussions explaining the powers and limits of Project5's algorithms, and Harpreet Singh of Sun Microsystems for his help in understanding and instrumenting `CallFlow` in the Glassfish server.

References

[1] Kruegel, C., Valeur, F., Vigna, G., Kemmerer, R.: Stateful intrusion detection for high-speed network's. In: IEEE Symp. on Security and Privacy (2002)
[2] Jiang, W., Song, H., Dai, Y.: Real-time Intrusion Detection for High-speed Networks. Computers & Security 24(4), 287–294 (2005)
[3] Krishnamurthy, B., Sen, S., Zhang, Y., Chen, Y.: Sketch-based change detection: Methods, evaluation, and applications. In: IMC 2003 (2003)
[4] Lakhina, A., Crovella, M., Diot, C.: Mining Anomalies Using Traffic Feature Distributions. ACM SIGCOMM Comput. Commun. Rev. 35(4) (October 2005)
[5] Mai, J., Chuah, C., Sridharan, A., Ye, T., Zang, H.: Is Sampled Data Sufficient for Anomaly Detection? In: IMC 2006 (2006)
[6] Barham, P., Donnelly, A., Isaacs, R., Mortier, R.: Using Magpie for Request Extraction and Workload Modeling. In: USENIX OSDI (2004)
[7] Chen, M.Y., Accardi, A., Kiciman, E., Lloyd, J., Patterson, D., Fox, A., Brewer, E.: Path-based failure and evolution management. In: USENIX NSDI (2004)
[8] Aguilera, M.K., Mogul, J.C., Wiener, J.L., Reynolds, P., Muthitacharoen, A.: Performance debugging for distributed systems of black boxes. In: ACM SOSP (2003)
[9] Reynolds, P., Wiener, J.L., Mogul, J.C., Aguilera, M.K., Vahdat, A.: WAP5: black-box performance debugging for wide-area systems. In: WWW 2006 (2006)
[10] Khanna, G., Varadharajan, P., Bagchi, S.: Automated online monitoring of distributed applications through external monitors. IEEE Trans. on Dependable and Secure Computing 3(2), 115–129 (2006)
[11] Khanna, G., Laguna, I., Arshad, F.A., Bagchi, S.: Stateful Detection in High Throughput Distributed Systems. In: SRDS 2007 (2007)
[12] The Java EE 5 Tutorial (September 2007), `http://java.sun.com/javaee/5/docs/tutorial/doc/`
[13] GlassFish: Open Source Application Server (2008), `https://glassfish.dev.java.net/`
[14] Klein, D., Manning, C.D.: Parsing with treebank grammars. Assoc. for Computational Linguistics (2001)

[15] Schuff, D.L., Pai, V.S.: Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface. In: IPDPS 2007 (2007)
[16] Kiciman, E., Fox, A.: Detecting application-level failures in component-based Internet services. IEEE Trans. Neural Networks 16(5), 1027–1041 (2005)
[17] Apache Tomcat: An Open Source JSP and Servlet Container, http://tomcat.apache.org/
[18] TPC-W Benchmark, http://www.tpc.org
[19] Grottke, M., Li, L., Vaidyanathan, K., Trivedi, K.S.: Analysis of Software Aging in a Web Server. IEEE Trans. on Reliability 55(3), 411–420 (2006)
[20] Brockwell, P.J., Davis, R.A.: Time Series: Theory and Methods, 2nd edn. (1998)
[21] Williams, A.W., Pertet, S.M., Narasimhan, P.: Tiresias: Black-Box Failure Prediction in Distributed Systems. In: IPDPS (2007)
[22] Laguna, I., Arshad, F.A., Grothe, D.M., Bagchi, S.: How To Keep Your Head Above Water While Detecting Errors. ECE Technical Reports, Purdue University, http://docs.lib.purdue.edu/ecetr/379
[23] Wu, Y.S., Bagchi, S., Singh, N., Wita, R.: Spam Detection in Voice-Over-IP Calls through Semi-Supervised Clustering. In: IEEE/IFIP DSN 2009 (2009)
[24] Rabiner, L.R.: A tutorial on Hidden Markov Models and selected applications in speech recognition. Proceedings of the IEEE 77(2) (February 1989)