4-1-2009

How To Keep Your Head Above Water While Detecting Errors

Saurabh Bagchi
Purdue University, sbagchi@purdue.edu

Ignacio Laguna
Purdue University - Main Campus, ilaguna@purdue.edu

Fahad Arshad
Purdue University, faarshad@purdue.edu

David M. Grothe
dgrothe@purdue.edu

How To Keep Your Head Above Water While Detecting Errors

Ignacio Laguna, Fahad A. Arshad, David M. Grothe, 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 the 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 and Hidden Markov Model (HMM)-based technique to select the messages and states that the detection system processes such that the non-determinism is minimized. We also present a mechanism for selecting computationally intensive rules to match based on the likelihood of detecting an error if a rule is completely matched. We demonstrate the techniques in a detection system called Monitor applied to a J2EE distributed online banking application. We empirically evaluate the performance of Monitor under different load conditions and compare it to a previous system called Pinpoint.

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 for errors 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 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 error detection as just *detection* in this paper.

Stateful detection is looked upon as a powerful mechanism for building dependable distributed systems [1][12]. 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 and increasing load from these 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* [12] which provides stateful detection by observing the messages exchanged between application components. Monitor is provided with a representation of the application (or protocol) behavior using a finite state machine (FSM) along with a set of normal behavior rules. A simple example of a rule for a three-tier e-commerce system is that a request submitted to authenticate a purchase request should complete within a user-specified time bound. Monitor uses an observer model whereby it observes the inter-component interactions, but does not have any knowledge of the internal state of a component. Monitor performs two primary tasks on observing a message. First, it deduces the application state by performing a state transition based on the observed message. Second, it performs rule matching for the rules associated with the particular state and observed message.

We have observed that Monitor has a breaking point in terms of the rate of messages it has to process [12]. 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 at the application level 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 [21]. A higher traffic rate exhausts its CPU and memory resources leading to packet losses and an increase of false alarms. For Monitor, we have observed through experimentation that the breaking point on a standard Linux box is 100 packets/sec [12].

We have shown in previous work [13] that we can reduce the processing load in Monitor 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 sense that the Monitor is no longer aware of the exact state the application is in. In sampling mode, messages are either sampled (and processed) or dropped. When a message is dropped, Monitor loses track of which state the application is in. This causes inaccuracies in selecting the rules to match since 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

We propose an *intelligent sampling* technique to reduce the non-determinism caused by random sampling in stateful detection systems. This technique is based on the observation that in an application's FSM, a message type can be seen as a state transition in multiple states. For example, a call to the *addressLookUp* function to look up the address information of an e-commerce site's consumer may be made from different components. Therefore, different message types differ in their ability to pinpoint which possible states the application is in. Computation of this discriminating property of the messages is done offline from the FSM of the application, which in turn is automatically generated through training traces. With intelligent sampling, Monitor observes all messages at runtime, but selectively samples and processes the messages with a high discriminating property, thereby limiting the non-determinism.

Even with the proper selection of messages, there is remaining non-determinism about the application state. Therefore, 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. This involves prior training of the model with representative application traces, a challenge but nevertheless used in many detection systems [5][8]. We show that the two techniques—intelligent sampling and HMM-based filtering—make Monitor scale to an application with a high load, with only a small degradation in detection quality.

For the evaluation, we use a distributed J2EE online banking application, the Duke's Bank application [15], running on Glassfish, the open source Sun Application Server [16]. Glassfish is comprised of a web container, an EJB container, and a back-end database. We inject errors in pairs of the combination (component, method), where 'component' can be a Java Server Page (JSP), servlet, or 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 [8] for detecting anomalies in the structure of web interactions and Monitor with random sampling [13].

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 we explain our experimental testbed and in Section 6, the experiments. In Section 7 we review related work and in Section 9 we present the conclusions and limitations of this work.

2 Background

In previous work we developed Monitor, a framework for online error detection in distributed applications [12]. Online implies the detection happens when the application is executing. Monitor is said to *verify* the application's components by observing the messages exchanged between the components. Monitor 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. This permits error detection to be applied to third party applications where source code is unavailable.

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—a Finite State Machine (FSM) and a set of application-level behavior-based rules. We define a *web interaction* as the set of inter-component messages that are caused by a 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 is provided a representation of the application behavior through an FSM that 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). Recent techniques for deriving the logic of low-level programs through an FSM have been proposed in [19]. In our current system, transitions are caused by method invocations on components and method returns. In addition, a behavior-based rulebase is provided to Monitor. Rules can be derived from the application specification or specified from QoS conditions required by the application's administrator. These rules can verify delays in components, or values of state variables. We explain more about rule types in Section 2.4 but the issue of how to generate appropriate rules is outside the scope of this paper.

When observing an application component's message, Monitor performs two primary tasks. First, it performs a state transition according to the FSM and the observed message. This allows Monitor to infer the current state of the application. Second, it matches rules associated with the particular state of the application and the observed message. If it is determined that the application does not satisfy a rule, an alarm is signaled.

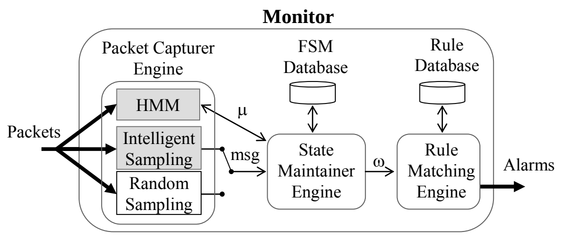

Fig. 1. Monitor architecture. One-sided and two-sided arrows show unidirectional and bidirectional flow of information respectively. Gray boxes indicate new components added to Monitor in this work.

Monitor architecture consists of three primary components, as shown in **Fig. 1** the PacketCapturer engine, the StateMaintainer engine, and the RuleMatching engine. Each component executes in a separate thread. Other components of the architecture are described in [12].

The `PacketCapturer` engine is in charge of capturing the messages exchanged between the application components, which could 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. In previous work [13] we showed that a *random sampling* approach helps in reducing Monitor's workload when experiencing high incoming rates of messages.

We define the term sampling here as we will use it for the rest of the paper. An incoming message into Monitor may be *sampled*, meaning, it will be processed further with the two important steps mentioned above (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, which makes it a lightweight operation. In [12] we have observed that under non-sampling conditions, Monitor's accuracy and precision suffer when the rate of incoming messages reaches 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 with a rate of one every $(R / (R - R_{th}))$ messages.

Sampled messages are passed to the `StateMaintainer` engine in order to perform state transitions according to the application's 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 since Monitor has an almost-complete view of the events generated in the application. Therefore it can infer the actual application's state, giving $|\omega|=1$. 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. In this scenario, ω ends up with a set of the possible states in which the application *can* be in, given the number of dropped messages. 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.

A challenge in Monitor, when performing random sampling, is to maintain high levels of accuracy and precision even while dropping messages. Two cases illustrate this degradation. First, if ω does not contain the correct application state S_i, and a failure occurs in S_i, Monitor will have a missed alarm since no rules will be applied to S_i. This leads to a reduction in accuracy. Second, if ω contains a large number of incorrect states—states where the application is not in—false alarms will increase.

Due to the randomness of the sampling approach proposed in [13], 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 by obtaining traces from the application when it 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*). The rationale for this level of granularity is to be able 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 within 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.

The FSM for Duke's Bank has 31 states and 62 events (2×31 because of calls and returns). 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 [5][8][22].

2.4 Rule Types

In previous work [12] 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. The response times can be arrived at by various means: (1) a protocol specification may specify that a component should acknowledge or reply to a request made from another component in a bounded interval of time, (2) a Service Level Agreement (SLA) may specify QoS constraints for web services or service components, (3) models based on performance analysis tools, such as Magpie [7], can be used to derive normal response time of elements in component-based applications. The other main type of rules we use is to verify that values of state variables lie within specified ranges, e.g., number of failed authentication attempts must not exceed a threshold.

3 Handling High Streaming Rates: Intelligent Sampling

3.1 Sampling in Monitor

With increasing incoming message rates at Monitor from the application components, its overall workload increases leading to higher latency of detection. To maintain an acceptable latency, Monitor chooses to process only a fraction of the incoming messages. When a message arrives at the `PacketCapturer` engine, a sampling mechanism is used to decide whether to pass the message to the `StateMaintainer` or not, i.e., whether to sample or to drop the message. If a message is sampled, it is then processed by the `StateMaintainer` to prune the state vector, and then by the `RuleMatching` engine for rule matching.

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 $\{S_B, S_D, S_E, S_F\}$. Depending on the number of consecutive dropped messages, the state vector can grow to a maximum of the total number of states in the FSM.

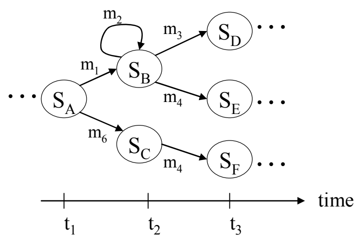

Fig. 2. A fragment of a Finite State Machine (FSM) to demonstrate non-determinism introduced by sampling.

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).

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 other appears 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 makes a difference to the resulting state vector. We say therefore that m_3 has a more *desired* 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*; 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, "call or return"). Sampling a message has the same definition as in Section 2.2, i.e., performing state transition and rule matching with the message.

Let us denote by d_m, *discriminative size*, 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 desired. The selection of such messages is implemented in Monitor through a greedy algorithm which we describe next.

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. This approach relies on a concurrent process that tracks the incoming message rate and triggers a recalculation of m and n when a significant change is detected.

Fig. 3. shows the `IntelligentSampling` algorithm. The algorithm runs for a window of m messages and the main while loop in lines 5-16 examines m messages, while the condition in line 6 guarantees that only n messages are sampled. If the last resort point is never reached (say the desired messages arrive early in the window), the algorithm always runs lines 7-11. Here discriminative size of messages is examined and those with $d_m < d_{th}$ are sampled. Each message is looked up in a pre-computed static table which has d_m values for all the messages. Otherwise, as a result of reaching the last resort point, it runs lines 12-14 where all messages are sampled.

```
IntelligentSampling decides whether to sample or drop a
message in a window of messages.
Input: n: the number of messages that have to be sampled; m:
size of the window of messages from which we sample n
messages; table: table with each message and its corresponding
discriminative size; d_th: threshold for the discriminative size of a
message.
Variables: currentMsg: current captured message; numMsgs:
number of messages seen in m; numSampled: number of sampled
messages; size: discriminative size of a message

IntelligentSampling(n, m, table, d_th):
1.  currentMsg ← getNextMessage( )
2.  numMsgs ← 0
3.  numSampled ← 0
4.  size ← 0
5.  while (numMsgs < m) then
6.    if ( numSampled < n ) then
7.      if ( n – numSampled < m – numMsgs ) then
8.        size ← discriminativeSize(currentMsg)
9.        if ( size < d_th in table) then
10.         SampleMessage(currentMsg)
11.         numSampled ← numSampled + 1
12.     else
13.       SampleMessage (currentMsg)
14.       numSampled ← numSampled + 1
15.   currentMsg ← getNextMessage( )
16.   numMsgs ← numMsgs + 1
17. return
```

Fig. 3. Pseudo-code for intelligent sampling algorithm.

The function `SampleMessage` passes the message to the `StateMaintainer` for further pruning of the state vector.

For a window of m messages, the computational cost of this algorithm is O(Km), where K is the cost of looking up the discriminative size of a message. We implemented *table* by using a hash-table so the expected time of this search is O(1), giving an overall complexity O(m). The space complexity is O(M), where M is the count of types of messages in the system.

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. In a subsequent stage, Monitor prunes states from the state vector that have low probability values.

An HMM [14] 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 [14] to estimate HMM parameters to model the Duke's Bank application. The HMM is trained with a set of traces from the application which is obtained by imposing a load of concurrent users for about 5 minutes. These are the same set of traces used to build the application FSM. We attempt to produce a complete list of all the web interactions that can occur in the application. The Baum-Welch algorithm starts with a uniform probability distribution for all states and edges and refines it using the traces.

4.2 Algorithm for Reducing the State Vector using HMM

We have implemented the `ReduceStateVector` algorithm (**Fig. 4**) 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).
- **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. For example, taking the example in **Fig. 2.**, if $\omega_t = \{S_B, S_D, S_E, S_F\}$ and $\alpha_t = [S_D, S_E, \ldots, S_A]$, by taking the top k=2 in α_t the new state vector ω_{t+1} would be $\{S_D, S_E\}$. Notice that 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. This step is executed in lines 5-11.

The time complexity of the algorithm is proportional to the time in computing $P(q_t = s_i \mid O, \lambda)$ 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 $O(N^3T + N \log N + Nk)$. In practice, the last factor tends to be N because we select a small constant for k (1 or 2).

`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 (this is the minimum size for the new state vector ω_{t+1}).
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($\lambda$, $O$, $\omega_t$, $k$):
1.  $\mu_t \leftarrow \varnothing$
2.  For each $i$ in $S$
3.          Add $P(q_t = s_i \mid O, \lambda)$ to $\mu_t$
4.  $\alpha_t \leftarrow sort(\mu_t)$ by $p_i$
5.  $I \leftarrow \varnothing$
6.  $I \leftarrow \omega_t \cap \alpha_t[1\ldots k]$
7.  if ( $I = \varnothing$ ) then
8.          $\omega_{t+1} \leftarrow \omega_t \cup \alpha_t[1\ldots k]$
9.  else
10.         $\omega_{t+1} \leftarrow I$
11. return $\omega_{t+1}$
```

Fig. 4. Pseudo-code for reducing state vector using HMM's estimate of probability of each application state.

Fig. 5 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).

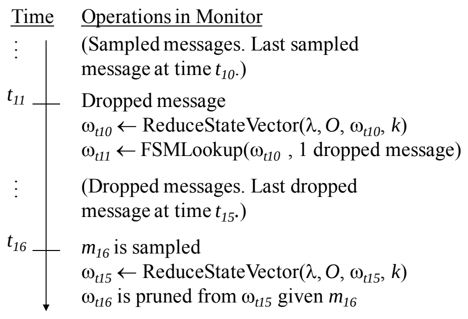
Time
Operations in Monitor
(Sampled messages. Last sampled message at time t_{10}.)
t_{11}
Dropped message
$\omega_{t10} \leftarrow$ ReduceStateVector(λ, O, ω_{t10}, k)
$\omega_{t11} \leftarrow$ FSMLookup(ω_{t10} , 1 dropped message)
(Dropped messages. Last dropped message at time t_{15}.)
t_{16}
m_{16} is sampled
$\omega_{t15} \leftarrow$ ReduceStateVector(λ, O, ω_{t15}, k)
ω_{t16} is pruned from ω_{t15} given m_{16}

Fig. 5. Example of points in time when the ReduceStateVector algorithm is invoked.

5 Experimental Testbed

5.1 J2EE Application

Many of today's distributed applications, such as e-commerce and online banking, are implemented using the J2EE standard. We use the J2EE Duke's Bank Application [15] as our experimental testbed. Duke's Bank provides user functionalities such as accessing account information and performing transactions, and is representative of a medium-sized component-based application. It is composed of 4 web components (servlets and Java Server Pages) and 6 Enterprise Java Beans (EJB) components. Duke's Bank is run on Glassfish v2 [16], the open-source application server from Sun Microsystems. 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 method, without needing any application change.

5.2 Web-users Emulator

To evaluate our solutions in diverse scenarios such as high user request rates and multiple types of workload, we wrote *WebStressor*, a web interactions emulator. WebStressor takes different user profile traces and replays them by sending each message in the traces to the tested detection systems. Each profile 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 has as configuration parameters: the number of concurrent users, the user think-time, and the ramp-up delay (the time between two successive users starting to interact with the system). For all the experiments, we wanted to create a specific number of users at a fast rate and impose a high load on the systems. Therefore, we used a random think-time between 1 and 5 seconds and a ramp-up delay of 200 milliseconds, which are relatively small values compared to say the TPC-W benchmark. WebStressor also has error injection capabilities which are explained in Section 6.5.

5.3 Pinpoint Implementation

Pinpoint [8] 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. We call this implementation Pinpoint-PCFG in the paper.

Pinpoint-PCFG has a training phase and an online detection phase. We use the implementation of Inside-Outside Algorithm (IO) from Brown University [17] for the estimation of the production probabilities to generate a stochastic CFG. Pinpoint-PCFG is trained using the same traces from Duke's Bank that are used to build the FSM and to train the HMM.

The online part of Pinpoint-PCFG is implemented using the Cocke-Younger-Kasami (CYK) parser algorithm [17], to parse the sequence of messages to determine the probability of deriving a web interaction. The probability of deriving a web interaction *j* from Pinpoint-PCFG is the product of the probabilities of productions used in the derivation. In our implementation, we apply a transformation to the raw probabilities to be able to work with small probability values. Let the web interaction *j* be derived through *n* productions and probability of production *i* be p_i. We calculate a measure M_j for the web interaction *j* by the following equation:

$$M_j = -\log \prod_{i=1}^{n} p_i \cdot$$

Note that, the higher the value of M_j for an observed web interaction, the lower is the probability of seeing that web interaction according to the PCFG model. We compute M_j for each web interaction encountered and compare it to a threshold (M_{th}); if M_j is greater then M_{th}, the interaction is marked as anomalous. In our experiments, we empirically determine the M_{th} value for optimal performance of Pinpoint-PCFG (Section 6.7).

6 Experiments and Results

In this section we report experiments to evaluate the performance of Monitor under different loads and for different types of injected errors. We also provide a comparative evaluation with the Pinpoint-PCFG algorithm. When we refer to Monitor, we mean baseline Monitor [12] with the 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 machine. We show 95% confidence intervals for some representative plots, but not all, to keep the graphs readable.

6.1 Measuring the Benefits of Sampling

Our first experiment is aimed to validate if sampling is needed in Monitor for practical user loads. We stress the application with a load of 20 concurrent users and measure the average rule matching time in Monitor-baseline and Monitor with sampling. Rule matching time is defined as the difference in time from when a message arrives into Monitor to when matching the rules corresponding to this message completes. The measurement is done every 2 seconds in a run of 5 minutes. **Fig. 6.**(a) shows the results for this experiment. In Monitor-baseline, the time increases considerably to the order of minutes as the experiment continues. The trend will be monotonically increasing since the queue is filled faster than it is being drained. We observe that with the sampling scheme, rule matching time is kept approximately constant. This empirically demonstrates the utility of sampling.

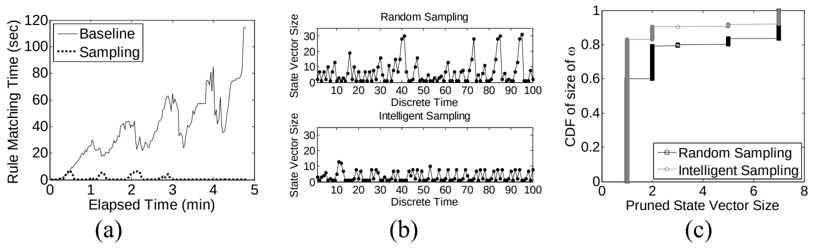

Fig. 6. Performance results when comparing Monitor baseline, random sampling and intelligent sampling. (a) Average rule matching time for Monitor baseline (without sampling) and Monitor with sampling with 20 concurrent users; (b) Sampled values of state vector ω for Monitor with random and intelligent sampling; (c) CDF for the pruned state vector ω with random and intelligent sampling.

6.2 Adjusting the Sampling Rate

When experiencing high incoming rates, Monitor starts sampling messages to avoid the latency exceeding a threshold L_{th}. Beyond a certain incoming rate, the latency in matching rules increases continuously from milliseconds to minutes and we would like the Monitor to operate at a rate less than where this occurs. We call this Monitor's *threshold rate* and denote it as R_{th}. We observe this threshold in Monitor when WebStressor emulates approximately 3 concurrent users in Duke's Bank application, which we use for the experiments as the trigger for sampling in Monitor. This corresponds to R_{th} of 52 messages/sec on average. Due to the artificially low think time and ramp-up time, 3 users in our experiments are equivalent to a higher number of actual users. Also 24 concurrent users define the upper end of our experimental range, not the range of capabilities for either Monitor or Pinpoint-PCFG.

Sampling rate R_s for an incoming rate R is given by:

$$R_s = \frac{R_{th}}{R}, R > R_{th}.$$

The performance of intelligent sampling is affected by the calculated sampling rate. The algorithm samples n messages in a window of m messages. Therefore, Monitor needs a fraction n/m that approximates R_s. We call this fraction the *sampling factor*. The discretization of the rate causes a discretization error for different number of concurrent users as shown in **Table 1.**.

The selection of m in the sampling factor is a critical step since it affects the performance of intelligent sampling. If m is greater than the size of a web interaction, intelligent sampling may, in pathological cases, drop all the messages in the web interaction. Then a fault in the web interaction will not be detected. Therefore, to achieve low missed alarms, the value for m should be small, say the size of the smallest web interaction, which is 6 for the Duke's Bank application. However a small size for m causes a large discretization error and curtails the ability of Monitor to select discriminating messages from within a large window. As a balance, we assign m=8 for the experiments. Table 1 shows the sampling factor values for different number of concurrent users, which maps to different message rates.

Table 1. Sampling Factors calculated according to the number of number of concurrent users

	Concurrent Users					
	4	8	12	16	20	24
Messages / sec.	70.00	118.97	194.28	236.86	314.38	362.74
Sampling rate	0.741	0.436	0.267	0.219	0.165	0.143
Sampling factor	6/8	4/8	2/8	2/8	1/8	1/8
Discretization error	0.01	0.06	0.02	0.03	0.04	0.02

6.3 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. This load results in a sampling factor 4/8. Monitor is run individually in random sampling (RS) mode, and in intelligent sampling (IS) mode. 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.

In each mode, we obtained 3337 sample values of ω's size. **Fig. 6.**(b) shows 100 snapshots of these values for RS and 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, allowing the `StateMaintainer` to produce smaller pruned state vectors ω. We notice that in RS mode, ω's size can reach 31 which is the number of states in the FSM, whereas in

IS mode, ω's size is bounded to 14. Also, ω's size can increase even when sampling is being done, but this happens less often with IS due to its ability to sample suitable messages.

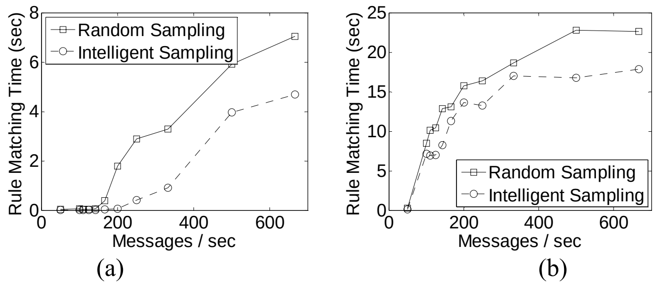

Fig. 7. Rule matching time for random sampling (RS) and intelligent sampling (IS) modes under sampling rates of 0.75 and 0.25 in figures (a) and (b) respectively.

We observe also in **Fig. 6.**(b) that for IS mode, ω's size is sometimes kept 1 consecutively for 4 messages. We can observe the occurrence of this pattern once in the range of samples 10−20, and two times in the range of samples 20−50. In the Duke's Bank application's FSM, the most frequent discriminative size, within the 62 types of messages, is 1. Therefore, as the intelligent sampling algorithm tries to sample these messages preferentially, it is likely to repeatedly attain ω of size 1. Our ultimate goal, regardless of the sampling algorithm, is ω of size 1 since the application in reality is in one state at any point in time.

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 for improved detection quality and latency. **Fig. 6.**(c) 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.

As more direct proof of the benefit of IS, we measure the average rule matching time for IS and RS modes. **Fig. 7.**(a)–(b)shows the average rule matching time under two sampling rate conditions: 0.75 and 0.25. The set of rules is fixed to 80%, and no error injection is performed. The HMM-based state vector reduction is also disabled to avoid including its overhead in these measurements.

The average rule matching time for Monitor with IS is lower than for RS for all cases. Even though IS incurs the additional overhead of determining the type of messages and looking up its discriminative size, the reduction of the state vector size more than compensates for it. For a sampling rate of 0.25, for IS the rule matching time remains negligible (< 1 sec) for incoming message rates less than 350 messages/sec. This supports the fact that Monitor's detection delay is in the order of milliseconds for up to 24 concurrent users (equivalent to 362.74 messages/sec).

6.4 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:

$$\mathit{Accuracy} = |C| / |I|; \mathit{Precision} = |C| / |D|$$

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.

It may be useful to identify a problem in a web interaction even before the web interaction finishes. For example, if during the sequence of calls between components in a web interaction, we detect a problem in a subcomponent, we may want to prevent subsequent subcomponents from being called. This can help in preventing the propagation of error to subsequent subcomponents and in diagnosing the root cause of the problem. If a detection system achieves detection before the web interaction completes, we say the system has a *pre-detection latency*, else we refer to it as a *post-detection latency*. In Pinpoint-PCFG, a complete web interaction *must* be observed to perform error detection, while in Monitor detection is performed without needing to look at the entire web interaction. Therefore, Pinpoint-PCFG always has post-detection, while Monitor may have pre- or post-detection.

6.5 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.

6.6 Detecting Performance Problems

For this experiment we inject delays to simulate performance problems in the set of 5 subcomponents listed in **Table 2.**.

Table 2. 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

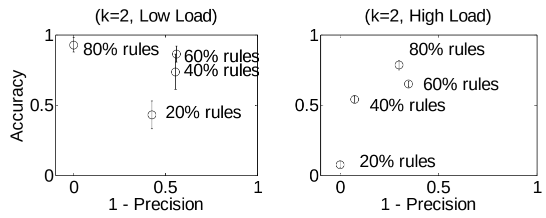

Fig. 8. ROC curves generated by varying the percentage of rules in Monitor.

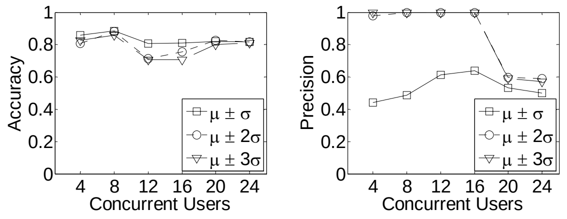

Fig. 9. Accuracy and precision of Monitor in detecting performance delays for three type of rules.

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 in **Table 2.** is touched in a web interaction.

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 and the imposed load of users to the application. We use two kinds of loads: low load (4 concurrent users), and high load (20 concurrent users). Once the ROC curves are generated, 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.

We vary two parameters for Monitor: the size of the rulebase and k in the HMM-based state vector reduction algorithm. Rulebase size is varied by activating randomly only 20%, 40%, 60% or 80% of the exhaustive list of rules, while k is varied over 1 and 2. **Fig. 8.** shows the ROC curves for k=2 (k=1 is not shown for lack of space). We observe that increasing the % of rules increases the accuracy both in low and high load, while we do not observe a consistent pattern in the variation of precision. The variation of k does not significantly affect the dispersion of points in the ROC curves. We select Monitor's best configuration parameters as rulebase size of 80% and k=2 and use it for the delay error experiments.

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: $\mu\pm\sigma$, $\mu\pm2\sigma$ and $\mu\pm3\sigma$.

Fig. 9. shows the results of this experiment. We observe that using $\mu\pm2\sigma$ provides the best combination of accuracy and precision. Rules of the type $\mu\pm\sigma$ provide the highest accuracy but poor precision, while rules of the type $\mu\pm3\sigma$ provide accuracy levels a bit less than $\mu\pm2\sigma$ and similar precision levels. For rule types $\mu\pm\sigma$ and $\mu\pm2\sigma$ 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. The decrease in precision happens since when the application is stressed with a high load, the response times increase naturally, causing Monitor to flag more false alarms. In future, we can improve precision by making the delay rules adapt to the number of concurrent users.

6.7 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.

For this experiment, first, we empirically determine the best value of parameter *k* for the HMM-based state vector reduction algorithm. **Fig. 10**(a) shows Monitor running with different values of *k* while we inject anomalous web interactions. Parameter k=0 represents Monitor running without HMM. We can observe that, with no HMM, in both low and high loads, accuracy is very low (about 0.4). For k=2 in low load and k=1 in high load, accuracy reaches its highest value. We observe that as we vary *k*, for low load, accuracy remains almost the same (0.9), whereas it decreases substantially in high load (reaching a minimum of about 0.55). This result validates our design that HMM is useful in detecting anomalous web interactions since Monitor with $k > 0$ performs better than with $k = 0$. 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, under high load, 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.

Second, we determine the best configuration parameter setting for Pinpoint-PCFG. We vary the threshold M_{th} to get Pinpoint-PCFG's ROC curves under low and high load. **Fig. 10**(b)–(c) show the results of this experiment. A lower value of threshold generates more false positives. A very high value on the other hand generates missed alarms. We select M_{th} = 350 as the operating point for Pinpoint-PCFG.

Fig. 10(d)–(e) show the results for accuracy and precision of Monitor and Pinpoint-PCFG. We observe that on average, Monitor's accuracy is comparable to 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. 10**(g)) while in Monitor it is in the order of milliseconds (**Fig. 10**(f)).

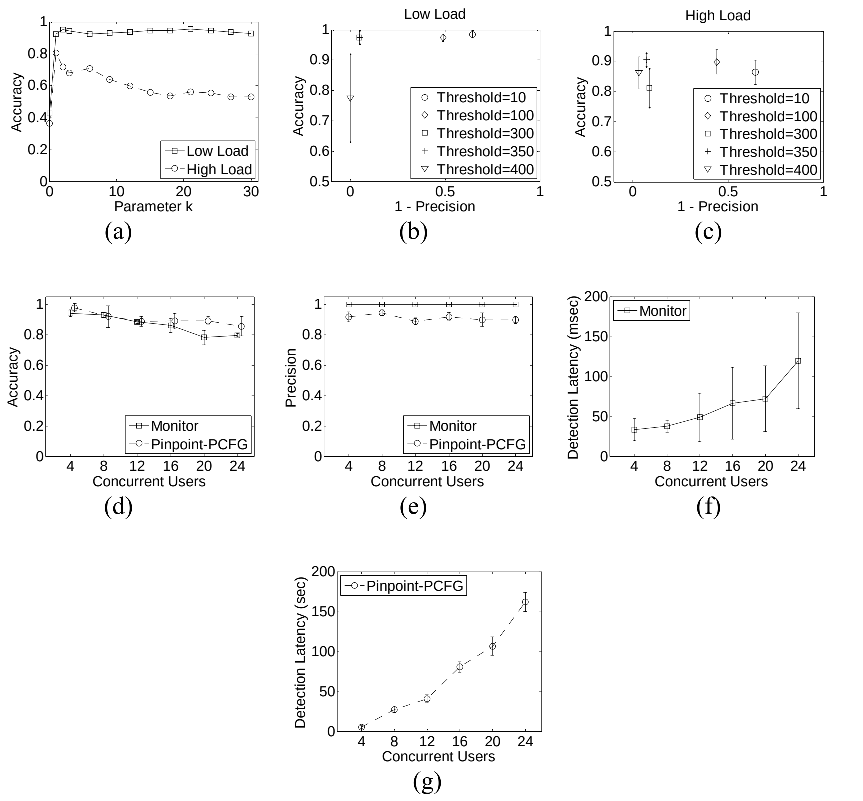

Fig. 10. Performance results for Monitor and Pinpoint when detecting anomalous web interactions. (a) Accuracy in Monitor when varying parameter *k* in the HMM-based state vector reduction algorithm; (b)–(c) ROC curves for Pinpoint-PCFG where points are generated by varying the threshold M_{th}; (d)–(e) Accuracy and precision for Monitor and Pinpoint-PCFG; (f)–(g)Detection latency for Monitor and Pinpoint-PCFG.

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.

An important property of a PCFG is that since the grammar is context-free, the PCFG represents a super-set of the observed web interactions seen in the traces used for training. Thus it can match some patterns that were not seen in the training phase. Still there are some patterns that are normal but since they were not seen in the training phase, generate a false alarm.

We observe that Monitor has a pre-detection latency fraction that varies from 8.93% to 22.67%, for the different numbers of concurrent users. Pinpoint-PCFG has only post-detection latency since it needs to see the entire web interaction before flagging an error.

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 [18] 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 3.**. This makes the Pinpoint-PCFG process thrash.

6.8 Fine-Grained Detection of Performance Problems

We 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. The rationale behind this changed definition is that IS is expected to instantiate a smaller subset of rules, relevant to the correct subcomponents, than RS. Thus, if our granularity of detection is the entire web interaction, then they may perform comparably—RS may flag a rule related to any of the touched subcomponents in the interaction, even a non-faulty subcomponent. IS, on the other hand, is more likely to flag a rule specifically for the faulty subcomponent. This difference can be brought out by defining accuracy and precision at the level of subcomponents. Detection at the level of a subcomponent this level is helpful in diagnosis—finding the root cause of the problem—since it helps in pinpointing suspect subcomponents.

Variables I, D, and C are now defined as: I: the subcomponents where faults were injected; D: the subcomponents in which Monitor detected a failure; C: out of I, the subcomponents in which Monitor detected a failure (these are the correct detections). Accuracy and Precision are defined (as in Section 6.3) as $|C|/|I|$ and $|C|/|D|$.

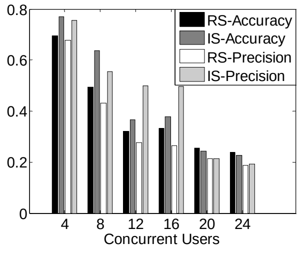

Fig. 11. Accuracy and Precision for Random Sampling and Intelligent Sampling for performance delay errors.

We inject delays in the same subcomponents used in Section 6.6, while varying the number of concurrent users. **Fig. 11.** shows the results of the experiment. 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.

6.9 Memory Consumption

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 3.** shows the results of this experiment. The size of a three-dimensional array for each web interaction encountered uses a large amount of virtual memory for Pinpoint-PCFG.

Table 3. 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

7 Efficient Rule Matching

In order to improve the performance of Monitor, we present a mechanism for efficient rule matching. As the load in Monitor depends on the processing overhead per incoming message, reduction in rule matching overhead will result in a reduction in Monitor's load. Rules that are computationally expensive are strong candidates for efficient matching since they cause Monitor to devote most of its resources to them.

Suppose that at time t, a message m_t is observed, a rule R has to be matched in the `RuleMatching` engine, 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. Our efficient rule matching mechanism works as follows:

1. Monitor estimates the probability P that, given the previous sequence of observed messages B, if R is matched, an error will be caught.
2. If P is greater than a threshold P_{th}, then R is matched. Otherwise, B becomes $\{m_{t-n+1}, \ldots, m_t\}$ and the `RuleMatching` engine goes to the step (1).

The challenge of this mechanism, given a rule R, is to estimate the model for P and the corresponding threshold value P_{th} that produce the best results in terms of catching an error, i.e., that maximize accuracy and precision of detection. While presenting a general theoretical approach for estimating the model and threshold for any rule is out of the scope of this paper, we present a simple example in which this technique can be used to efficiently detect a memory leak in the Apache Tomcat web server[23]. Apache Tomcat is an open-source web server written in Java that implements the Java Servlet and the JavaServer Pages (JSP) technologies from Sun Microsystems.

7.1 Memory Leak Injection

We instrumented the Apache Tomcat web server source code to inject a memory leak. Upon receiving a legitimate request, an unused object is created with probability p_{leak}, and it is kept referenced while the server runs so that it is not taken by the Java garbage collector. The result is an increase of memory usage that can be observed from the Java process running the server.

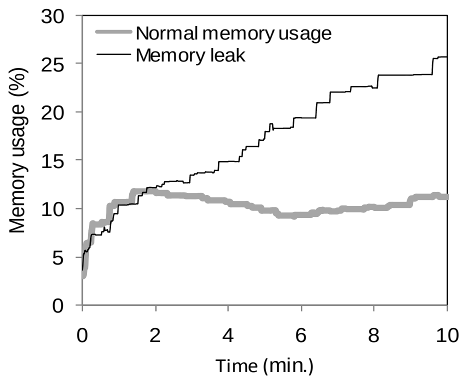

Fig. 12. Percentage of memory usage of the Apache Tomcat web server under normal conditions and with a memory leak fault injection.

We perform experiments to observe the pattern of the memory consumption of the web server in both in normal conditions and when the memory leak is injected. We use a test-bet of an e-commerce site that simulates the operation of an online store according the TPC-W benchmark[24]. 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. 12** 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 on average is imposed. Measurements are taken in a fixed interval of 1 second for a window of 10 minutes after the server is started.

7.2 Modeling Memory Consumption

Previous work on software rejuvenation[25] has proposed the use of time series analysis to model memory usage patterns in the Apache web server. In software rejuvenation techniques, time series analysis is used to understand aging and to predict when to reboot the server in order to avoid future failures. In this paper, we use time series analysis to build rules that are able to pinpoint a memory leak and that help us to demonstrate our efficient rule matching technique. 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[26]:

- 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[27]. In order to estimate the number of p and q parameters that best fit the models, whiling trying to keep these numbers small, we vary p and q over 1, 2, and 3. We then select the number 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 our test-bed, p=3 and q=2 resulted in the best configuration for the models.

7.3 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. When the rule is matched at runtime, it takes as input a sequence *B* of *n* new observed messages from the Apache Tomcat web server in which it will look for errors. Then, two simulated sequences *S* and *S'* are generated from the two models λ and λ'respectively using previous observed data values, and they are compared to *B* by measuring the RMS error. If *B* fits better in *S'*, i.e., its RMS error is less than the one for *S*, an error is flag by the rule indicating a possible trend of high memory consumption.

To observe the effectiveness of selectively matching this rule we use a simple mechanism for evaluating *P*, the probability of catching the error if the rule is matched. We use indication of instability in the system as a mechanism for having a meaningful value of *P*. We measure the standard deviation σ of the *m* previous observed values of memory consumption, and if it is greater than a threshold P_{th}, then the rule is matched. Notice that, in order to observe a reduction on the workload of Monitor, the overhead in evaluating *P* has to be less than the overhead of evaluating the whole rule; otherwise it may me better matching the rule directly.

Table 4. Detection coverage and average rule matching delay the ARMA-based rule.

Rule Matching Criteria	Memory Leak Detected	Average Matching Latency (msec)
Allways matched	yes	19.2835
$\sigma \geq 0.5$	yes	7.1148
$\sigma \geq 1.0$	no	1.25

Table 4. Detection coverage and average rule matching delay the ARMA-based rule. shows the results for 3 different configurations in Monitor when the memory leak is injected in the web server: (1) the rule is always matched, (2) the rule is matched only if $\sigma \geq 0.5$ in the last *m* messages, and (3) the rule is matched only if $\sigma \geq$ 1.0 in the last *m* messages. For the three experiments, $m = 5$ and the same workload that we used for training is imposed to 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.

As presented in table 4, in these experiments we observe whether the rule flags the memory leak, and the average *rule matching latency* in Monitor, which is defined as the time spent by Monitor to provide an *answer* on each incoming message, where the answer can be either the system is operating normally or an error is detected. 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 not be caught as it is the case when σ=1.0.

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 [8]. 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 6.7. A variant of the Pinpoint work [22] 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 [9][10] and Magpie[7]. 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 [9] 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 [10] 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 6.6).

The Magpie project [7] 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 models can be used for capacity planning, performance debugging, and anomaly detection. The workload models of request behavior can be used in Monitor to specify rules for detection of performance bottlenecks.

Other powerful tools have also been proposed recently. For example, in [11] authors present a tool called *liblog* that aids in recreating the events that occurred prior to and during failure.

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. This work looks for problems at the network level while we look at application level deviations from expected behavior.

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][5], for example, for detecting denial-of-service (DoS) attacks or worm scans. However, some studies show that these sampling techniques introduce fundamental bias that depredates performance when detecting network anomalies (e.g., in [6]). Our work matches rules based on aggregated information at the application level, while this work matches rules based on network level 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 mechanism for efficiently matching rules that can be computationally expensive based on the observation that rules will more likely catch errors if instability in the system is observed.

A disadvantage of our HMM-based technique is that an application with a large number of states can make the HMM processing too expensive. For the Duke's Bank application with 62 states, use of the HMM is beneficial as evidenced by the improvement in detection precision. Even when our HMM takes as input complete sequences of messages, the computational cost of this is less than sampling all the messages. 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 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 flagging it since rules associated with that state would not be instantiated. 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 will point to erroneous conditions in the application.

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] C. Kruegel, F. Valeur, G. Vigna and R. Kemmerer, "Stateful intrusion detection for high-speed network's," IEEE Symp. on Security and Privacy, 2002.
[2] W. Jiang, H. Song and Y. Dai, "Real-time Intrusion Detection for High-speed Networks," Computers & Security, vol. 24, Issue 4, Jun 2005, pp. 287-294.
[3] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," IMC 2003.
[4] P. Barford, J. Kline, D. Plonka, and A. Ron, "A Signal Analysis of Network Traffic Anomalies," IMC 2002.
[5] A. Lakhina, M. Crovella and C. Diot, "Mining Anomalies Using Traffic Feature Distributions," ACM SIGCOMM Comput. Commun. Rev., vol. 35, issue 4, Oct 2005.
[6] J. Mai, C. Chuah, A. Sridharan, T. Ye and H. Zang, "Is Sampled Data Sufficient for Anomaly Detection?," IMC 2006.
[7] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for Request Extraction and Workload Modeling," USENIX OSDI, 2004.
[8] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-based failure and evolution management," USENIX NSDI, 2004.
[9] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," ACM SOSP, 2003.
[10] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera and A. Vahdat, "WAP5: black-box performance debugging for wide-area systems," WWW 2006.

[11] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay Debugging for Distributed Applications," USENIX Annual Technical Conference, 2006.
[12] G. Khanna, P. Varadharajan and S. Bagchi, "Automated online monitoring of distributed applications through external monitors," IEEE Trans. on Dependable and Secure Computing, vol.3, no.2, pp.115-129, April-June 2006.
[13] G. Khanna, I. Laguna F. A. Arshad and S. Bagchi, "Stateful Detection in High Throughput Distributed Systems," SRDS 2007.
[14] L.R. Rabiner, "A tutorial on hidden Markov models and selected applications in speech recognition," Proc. of the IEEE, vol.77, no.2, pp.257-286, Feb 1989.
[15] The Java EE 5 Tutorial. http://java.sun.com/javaee/5/docs/tutorial/doc/, Sep 2007.
[16] GlassFish: Open Source Application Server. https://glassfish.dev.java.net/, 2008.
[17] Open Source Software Written by Mark Johnson. http://www.cog.brown.edu/~mj/Software.htm, Mar 2008.
[18] D. Klein and C. D. Manning, "Parsing with treebank grammars," Assoc. for Computational Linguistics, 2001.
[19] N. Kothari, T. Millstein, R. Govindan, "Deriving State Machines from TinyOS Programs Using Symbolic Execution," IPSN 2008.
[20] A. Dan et al, "Web Services on Demand: WSLA-driven automated management," IBM Systems Journal, vol 43(1), pp.136-158, 2004.
[21] D. L. Schuff, V. S. Pai, "Design Alternatives for a High-Performance Self-Securing Ethernet Network Interface," IPDPS 2007.
[22] E. Kiciman, A. Fox, "Detecting application-level failures in component-based Internet services," IEEE Trans. Neural Networks, vol.16, no.5, pp.1027-1041, Sept. 2005.
[23] Apache Tomcat: An Open Source JSP and Servlet Container. http://tomcat.apache.org/.
[24] TPC-W Benchmark. http://www.tpc.org
[25] M. Grottke, L. Li, K. Vaidyanathan, and K. S. Trivedi, "Analysis of Software Aging in a Web Server," IEEE Trans. on Reliability, Vol. 55, No. 3, pp. 411-420, 2006.
[26] P. J. Brockwell, R. A. Davis, "Time Series: Theory and Methods," Second Edition, 1998.
[27] The R Project for Statistical Computing. http://www.r-project.org.