7-7-2008

Online Error Detection for High Data Rate Distributed Applications

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

PURDUE UNIVERSITY
GRADUATE SCHOOL
Thesis/Dissertation Acceptance

This is to certify that the thesis/dissertation prepared

By Ignacio Laguna

Entitled Online Error Detection for High Data Rate Distributed Applications

For the degree of Master of Science in Electrical and Computer Engineering

Is approved by the final examining committee:

S. Bagchi Chair	
S. Rao	
J. P. Allebach	

To the best of my knowledge and as understood by the student in the *Research Integrity and Copyright Disclaimer (Graduate School Form 20)*, this thesis/dissertation adheres to the provisions of Purdue University's "Policy on Integrity in Research" and the use of copyrighted material.

Approved by Major Professor(s): S. Bagchi

Approved by: M. J. T. Smith 7/3/08
Head of the Graduate Program Date

Graduate School Form 20
(Revised 10/07)

PURDUE UNIVERSITY
GRADUATE SCHOOL

Research Integrity and Copyright Disclaimer

Title of Thesis/Dissertation: Online Error Detection for High Data Rate Distributed Applications

For the degree of Master of Science in Electrical and Computer Engineering

I certify that in the preparation of this thesis, I have observed the provisions of *Purdue University Executive Memorandum No. C-22,* September 6, 1991, *Policy on Integrity in Research.**

Further, I certify that this work is free of plagiarism and all materials appearing in this thesis/dissertation have been properly quoted and attributed.

I certify that all copyrighted material incorporated into this thesis/dissertation is in compliance with the United States' copyright law and that I have received written permission from the copyright owners for my use of their work, which is beyond the scope of the law. I agree to indemnify and save harmless Purdue University from any and all claims that may be asserted or that may arise from any copyright violation.

Ignacio Laguna
Signature of Candidate

7/2/2008
Date

*Located at http://www.purdue.edu/policies/pages/teach_res_outreach/c_22.html

ONLINE ERROR DETECTION FOR HIGH DATA RATE DISTRIBUTED APPLICATIONS

A Thesis

Submitted to the Faculty

of

Purdue University

by

Ignacio Laguna Peralta

In Partial Fulfillment of the

Requirements for the Degree

of

Master of Science in Electrical and Computer Engineering

August 2008

Purdue University

West Lafayette, Indiana

This work is dedicated to my parents who have gave me nothing but all the support and love I could ask for achieving my goals in live.

ACKNOWLEDGMENTS

Special thanks to Prof. Saurabh Bagchi for his guidance while pursuing my work; his support, patience and motivation have been invaluable for me in these years as a graduate student. Thanks also to Prof. Jan P. Allebach and Prof. Sanjay Rao for their time in serving on my committee.

I want to thank as well the other members of my research group, Gunjan Khanna, Fahad A. Arshad and David M. Grothe, who have gave me precious help for accomplishing this work.

TABLE OF CONTENTS

LIST OF TABLES

LIST OF FIGURES

ABSTRACT

Laguna Peralta, Ignacio. M.S.E.C.E., Purdue University, Ago 2008. Online Error Detection for High Data Rate Distributed Applications. Major Professor: Saurabh Bagchi.

Distributed systems comprising interacting services need runtime error detection to catch errors arising from software bugs, hardware errors, or unexpected operating conditions. A significant class of detection systems performs detection at the application level, based on the state of the application. For example, rule-based systems match rules against the application's state deduced by the detection system at runtime. Many large-scale distributed applications generate a high rate of messages which can overwhelm the capacity of the detection system. An approach to handle this is sampling, that is, processing only a fraction of the messages. 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 inaccuracies in matching state-based rules causing degradation in the quality of detection. In this work, we present an approach to select the messages to sample and process such that the non-determinism is minimized. Next, we present a Hidden Markov Model-based technique to probabilistically identify which application states are most likely so that the detection system can perform rule-based detection for only those states. We demonstrate the techniques in a detection system called Monitor applied to a Java-based three-tier online banking system. The techniques do not need application modifications or a priori application model, but do require knowledge of expected application behavior to come up with the rules. We empirically evaluate accuracy and precision of detection under different load conditions and compare our solution with two other state-of-the-art systems: Pinpoint and Convolution algorithm.

1. INTRODUCTION

1.1 Failure Detection in Distributed Systems

Increased deployment of high-speed computer networks has made distributed systems ubiquitous in today's connected world. The Internet Domain Name System (DNS), e-commerce and online banking systems, *Skype*, and web services providers such as *Google* and *Amazon.com*, constitutes some examples of the backbone of the IT infrastructure in the world today. We increasingly face the challenge of failures due to natural errors and malicious security attacks affecting these systems. Downtime of a system providing critical services such as in power systems, space flight control, banking and railways signaling could be catastrophic. It is therefore imperative to build detection systems that can detect problems with low latency—the time that elapses between the failure and the detection should be short. Low-latency detection enables diagnosis and recovery phases to take place before user services are impacted significantly, thus preventing long system downtime.

There are several challenges to the problem of designing a detection system that can handle failures in the distributed systems of today. For example, many existing systems run legacy code, the protocols have hundreds of participants, and systems often have soft real-time requirements. A common requirement for the detection system is to be *nonintrusive* to the observed system. This implies that significant changes to the observed application or to the environment in which it executes are undesirable. This also rules out executing heavyweight detectors in the same host as the application's components. While it may be possible to devise optimized solutions for individual distributed applications, such approaches are not very interesting from a research standpoint because of limited generalizability. Trying to make changes to a particular protocol or application also requires an in-depth understanding of the code. The application may be closed source, or even if open source, adequate expertise to understand and modify the code may not be readily available.

To address the challenge of being nonintrusive to the observed system, recent work from several research groups treats the application system's components as *black-box components*, that is, the observed system is viewed as a collection of software components typically without the source code available and with no (or minimum) knowledge of their implementation. Normal-behavior models are then built up so that failure detection and diagnosis is performed by looking at deviations from the models. For example, in [1] authors propose algorithms to extract component's response delay from traces obtained during a training phase while user requests travel thought a distributed e-commerce application. Faulty components that cause high latency to the system can be identified by comparing their response delay to those learned when observing the traces. In techniques like this, measurements needed for detection, such as component's response delays, are typically obtained by observing the interaction between the system's components which can be in the form of network messages or intra-node messages. When this is implemented so that no changes to the application are required to observe massages, the approach is called *black-box instrumentation*. Examples of modeling techniques based on black-box instrumentation that can be used for failure detection are Magpie [2][3] and Pinpoint [4].

In prior work, we have developed Monitor [5], a failure detection system for distributed applications. Monitor detects failures under the principle of black-box instrumentation—detection is performed by observing the messages exchanged between application's components. Monitor is provided a representation of the application or protocol behavior using a Finite State Machine (FSM) along with a set of normal behavior rules. An example of a rule can be that, in a three-tier e-commerce system, if a request submits a read command to the back-end database server, the command should complete within an administrator-specified time bound. When observing a message, Monitor performs two primary tasks. First, it deduces the state of the application and then performs a state transition in the application's FSM based on the observed message. Second, it performs rule matching for pre-specified rules. The rules are stateful rules, so that they are associated with a combination of the deduced state and the observed message. Monitor uses an observer model whereby it does not have any information about the internal variables of the application's components at runtime, but only observes the inter-component interactions. In contrast to [1] where detection is performed offline by looking at application's traces, the detection in Monitor is performed in an online manner as the application executes.

1.2 Scalability Challenges in High Throughput

Failure detection can broadly be classified as stateless detection and stateful detection. In the former, detection is done on individual messages by matching certain characteristics of the message, such as finding specific signatures in the payload. A more powerful approach is the stateful approach, in which the failure detection system builds up a *state* related to the application by aggregating multiple messages. Rules are then applied based on the application's state; thus on *aggregated information* rather than *instantaneous information*. Monitor follows the characteristics of a stateful rule-based detection system in which rules characterize normal behavior in the system. Stateful detection is widely recognized as being more powerful in its detection capability compared to stateless detection [6][7][8].

Though the merits of stateful detection are well accepted, scaling a stateful detection system with increasing number of application components, number of users or increasing data rate is a challenge. This is due to the increased processing load of tracking the application state and rule matching. The rules can be heavy-duty and can impose high load for matching. For example, a rule may be used to validate data consistency through different tables in a database. If an entry in a table is deleted, a consistency check has to be performed so that all associated entries have been garbage collected. This consumes significant computing resources. It may be tempting to think that throwing more computational resources would solve the problem. However, it is desirable to keep the resource requirements of the fault tolerance infrastructure smaller than that of the actual application system. Also, it would be advantageous to have the fault tolerance infrastructure scale up better as the application size increases. Therefore the stateful detection system has to be designed such that its resource usage is minimized.

We have observed in prior work [5] that Monitor has a *breaking point* in terms of the incoming message rate; beyond this point, its accuracy and latency of detection suffer. We observe through a testbed experiment that as the incoming packet rate into a single Monitor is increased beyond 100 packets/sec, Monitor breaks down on a standard Linux box. In this testbed, Monitor was configured to detect failures in a reliable multicast protocol called TRAM. The drop in accuracy or rise in latency can be very sharp beyond the breaking point depending on the exact queuing mechanism employed in Monitor. Note that *all* detection systems that fall in this important class of stateful detection, are expected to have such a breaking point, though the rate of messages at which each system breaks will be different.

Consider that the processing load on the detection system per unit time is given by (*the number of components being verified*) × (*the external message rate generated by each component*) × (*the processing overhead for each message*). We can reduce the processing load in Monitor by *sampling* incoming messages, that is, by processing only a fraction of them. This is complementary to the effort at making the per-message processing extremely efficient, which we have done and reported in [9]. Based on this fact, we implemented in prior work a *random sampling* approach to help Monitor reduce its workload when experiencing high incoming rates of messages [9]. When Monitor perceives a rate of incoming messages close to the breaking point, it activates a sampling-and-dropping mechanism for incoming messages to reduce the workload of state transition and rule matching. Messages are sampled randomly without looking at the content of the message. The sampling rate is such that the rate of messages being processed is well below the Monitor's breaking point.

1.3 Main Contributions of this Work

Because Monitor deduces the application's state based on received messages, when performing sampling of the messages, Monitor is no longer aware of the exact state the application is in. This phenomenon is called *non-determinism*. Monitor builds up a set of states that represents the possible states the application can be in, given a count of dropped messages. This causes inaccuracies in selecting the rules to match since the rules are based on the application state and the observed message. Inaccuracies in the selection of the rules to match lead to lower accuracy and precision (false and true positive rates) for Monitor.

In this work, we focus in reducing the non-determinism and increasing accuracy and precision of detection when Monitor is performing sampling. We show that the non-determinism can be bounded to a low value by selecting which messages to sample and process. This idea arises out of the observation that in an application, a message type can be seen in the transitions for multiple states within an FSM. For example, a call to the *getCustomerData* function to look up customer related information in an e-commerce site may be made from different components. Different message types differ in their ability to discriminate the application's current state, that is, to pinpoint which possible state(s) the application may be in. Such computation of the discriminative property of the messages is done offline by analysis of the application's FSM. At runtime, the Monitor observes all messages, but selectively samples and processes the messages with a high discriminative property. We call this technique *intelligent sampling*.

Even with the proper selection of messages during sampling, there is remaining non-determinism about the application state. Next, we use a Hidden Markov Model (HMM)-based technique to estimate the likelihood of the different application states. The HMM can answer the question—given a certain sequence of observations, what is the probability that the application is in any given state. Rule matching is then performed for a reduced set of states containing only the more likely states. This involves prior training of the HMM with representative application traces. This is admittedly a challenge but is nevertheless used in many systems of this genre, for example in [10]. The HMM enables Monitor to detect anomalies in the structure of request paths in the application, for example, request paths that results from malfunctioning components.

The two techniques (intelligent sampling and HMM-based estimation of the probable application states) are applicable to the general class of detection systems that relies on rule matching based on the application state. We believe that the presented techniques can make detection systems scale to large-size applications or to applications that exhibit high rates of messages, with only a small degradation in the detection quality. In this work, we incorporate the techniques in the Monitor system and experimentally quantify the detection quality in terms of accuracy and precision for a range of software errors.

Our experimental application is called Duke's Bank [11], a medium-size online banking application running on Glassfish, the open source Sun Application Server. Glassfish is comprised of a web container, an EJB container, and a back-end database. When a user request is received in the application server, Monitor observes the sequence of calls and returns between the application's components caused by the request. We call this sequence of calls and returns a *web interaction*. We inject errors in particular pairs of the combination (*component*, *method*), where component can be a Java *jsp*, *servlet* or Enterprise Java Bean (EJB), and method a function call in the component. The injected errors can cause failures in the combination (*component*, *method*), or in web interactions 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 back to the user.

We compare Monitor's performance with two state-of-the-art detection techniques: Pinpoint [4] and the Convolution algorithm from [1]. Pinpoint can detect anomalies in the structure of web interactions, while Convolution can determine unusually long delays in interactions between multiple components. While evaluating these three systems, we create a load that is imposed to the Duke's Bank by varying the number of concurrent users. We show that Monitor has a comparable accuracy to Pinpoint but a better precision when detecting anomalies in web interactions. We do not have final results in the comparison with Convolution algorithm (since its implementation is still in progress), but preliminary results show that accuracy and precision in Monitor is superior to that of Convolution. Monitor outperforms Pinpoint in terms of latency of detection by providing an average latency in the order of milliseconds while Pinpoint shows an average in the order of seconds.

2. BACKGROUND

In previous work, we developed Monitor, a framework for detecting errors in distributed applications [5]. Monitor is said to *verify* the application's components by observing the messages exchanged between the components through the communication channel. Monitor uses the observed messages and an application's FSM to deduce the state of the application. Next, pre-specified rules in a rulebase are used to verify correctness of the behavior based on the FSM. The rules can be either derived from the application specification (e.g., a protocol specification) or specified by the system administrator (e.g., constraints to meet QoS requirements).

The major futures and advantages of Monitor can be depicted as follows. First, Monitor performs detection through black-box instrumentation, that is, the application being verified does not have to be changed to allow Monitor detects errors—only exchanged messages are observed in the application's components, but not the internal variables of the application. This permit error detection to be carried out in an online manner, allowing faster detection latency as compared to offline approaches. Since Monitor operates asynchronously to the verified application, it does not become a performance bottleneck in the application. Second, a hierarchical structure has been implemented so that Monitor scales with large number of components or application entities located in different locations in the network. Third, Monitor is applicable to a large class of applications with minimal effort in moving from one application to another. This is achieved by keeping Monitor architecture application neutral and the rulebase specified in an intuitive formalism that can be used for detection of failures in different applications. Also the set of detectable errors can be extended in a modular manner without changes to either the application or the Monitor's algorithms. Finally, Monitor ought to have a low latency of detection since high latency will make any subsequent diagnosis, containment, or recovery complicated.

2.1 Fault Model

Monitor can detect any error that manifests itself as a deviation from the application's model and expected behavior that are given as input—a Finite State Machine (FSM) and a set of application-level normal-behavior rules. This is performed in Monitor without looking at the internal application variables and states. In the context of component-based web applications, an FSM is used to pinpoint deviations or anomalies in the structure of the observed web interactions, while rules are used to determine deviations from the expected normal behavior of application's components. A fault in an application component, for example a performance problem, can be manifested in a web interaction through a delayed response to a client request.

In Monitor, error detection can help in diagnosis because rules can be applied at the level of methods (or function calls) in components—a finer granularity than detecting an error in an entire web interaction or only a component.

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). (An example of a recent technique to deriving the logic of low-level programs through an FSM has been proposed in [12].) In our current system, we analyzed sequences of function calls obtained in application traces to derive a simple message-driven FSM. In addition, rules provided to Monitor can be derived from the application specification or specified from quality-of-service conditions required by the application's administrator. We explain in detail rule types and syntax in subsection 2.4.

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 to 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.

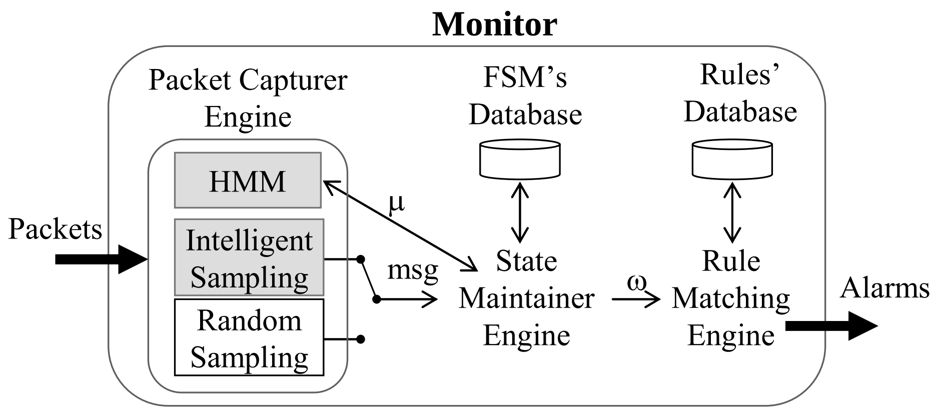


Fig. 2.1. Monitor architecture. One-sided and two-sided arrows show unidirectional and bidirectional flow of information respectively. Grey boxes indicate new components added to Monitor in this work

Monitor architecture consists of three primary threads as shown in Fig. 2.1: the PacketCapturer engine, the StateMaintainer engine, and the RuleMatching engine. Other components of the Monitor architecture are described in [5]—in this work we only explain the necessary components to present our novel ideas.

The PacketCapturer engine is in charge of "capturing" the messages exchanged between the application components. 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-and-dropping mechanism to reduce the workload of state transition and rule matching. In previous work [9] we showed that a random sampling approach helps in reducing Monitor's workload when experiencing high incoming rates of messages. In random sampling, messages are sampled randomly without looking at the content of the message—this approach is agnostic to the type of messages coming in, which prevents Monitor having to decide which message to drop or to keep. In [5] we have showed that under non-sampling conditions, Monitor's accuracy and precision suffers 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 close or > R_{th}, in which Monitor drops messages uniformly with a rate of one every (R / (R – R_{th})) messages.

The two gray boxes in Fig. 2.1, a Hidden Markov Model (HMM) and an intelligent sampling algorithm, correspond to the two new components we incorporate in Monitor as part of this work. We will explain the functionality of these two blocks later in section 3 and 4 respectively.

Captured 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 producing a representation of the application state which we call the *state vector* and it is represented by ω. When Monitor is in non-sampling mode, the state vector contains typically only one state—Monitor has an almost-complete view of the events generated in the application, therefore it can infer the actual application's state. However, when sampling mode is activated, Monitor loses track of the actual state of the application since it is not observing every event (or message) 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_i is sampled, ω is updated according to the states in ω and m_i. This is performed by observing (in the FSM) the new state (or states) to where the application could has been moved, from each state in ω given m_i. We call this mechanism *pruning* of the state vector and it is explained in more detail in subsection 3.1. Typically, when ω is pruned, its size is reduced, or tends to be 1, since in reality the application can only be in one state. However, for particular FSMs, pruning ω can increase its size, for example, when m_i appears in transitions to different states in the FSM, and initial states of these transitions are elements of ω.

The RuleMatching engine is responsible for obtaining the current (and pruned) state vector ω from the StateMaintainer engine and matching rules associated with the state(s) in ω. If the application is found to violate any of the rules, alarms are generated indicating that a problem exists in the application, so further actions can be taken—a diagnosis phase can be triggered either in Monitor or in a separate system in order to detect the root cause of the problem.

A challenge in Monitor when performing random sampling is to maintain high levels of accuracy and precision of detection while dropping messages, that is, while the view of the generated application's events is reduced. In Monitor, stateful detection is performed by matching normal-behavior rules per state; therefore, the degraded view of the application's state caused by random sampling affects directly its quality of detection. Typically two cases can illustrate this degradation when ω is produced through any sampling mechanism. First, if ω does not contain the correct application's state S_i, and a failure occurs in S_i, we 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 can be triggered leading to a reduction in precision. We have observed that due to the randomness of the sampling approach proposed in [9], these two cases can take place frequently enough to take down quality of detection to non-acceptable ranges. For example, in [9] we obtained a maximum accuracy of 0.7 when detecting failures in TRAM, a reliable multicast protocol. Systems running critical services can demand higher levels of accuracy and lower detection latency.

2.3 Building FSM from Traces

Monitor uses a stateful model (an FSM) for rule matching by preserving the state of the application across the received messages. This FSM is typically obtained from the protocol or application specification and it is given to Monitor as an input. However, we want our techniques to be applicable to a broader spectrum of systems, that is, systems in which the source code is not available and in which there is no knowledge of the internal behavior of components. For this purpose, an FSM of the Duke's Bank Application is built by obtaining traces from the application and observing the interaction between components, that is, their calls and returns between component's methods.

When we generate application's traces, no error injection is performed and we assume that the application is free of faults. This has also been assumed when performing training in other detection techniques such as in Pinpoint [4], our point of comparison in this work. A disadvantage of this assumption is that Monitor is not able to detect design faults that are not manifested as failures when obtaining the traces. However, if an FSM is provided from the application or protocol specifications, this problem is not present.

A state S_i in the FSM is defined as a tuple (*component*, *method*) where *component* is an Enterprise Java Bean (EJB), a servlet or a jsp web component, and *method* is a Java method within the component. In the rest of the paper we use the term *subcomponent* to denote the tuple (*component*, *method*). The aim for this level of granularity is to be able to pinpoint performance problems or errors in particular methods, rather than only in components.

The FSM is represented as a state transition diagram, in which a transition from a state S_i to another state S_j is enabled when a condition is satisfied. For example, (*AccountControllerBean*, *getDetails*) and (*AccountControllerBean*, *find*) constitutes two different states in our FSM, although they represent two methods in the same component (or class). A condition for a state change is defined as a *call* or *return* event, that is, the event that either subcomponent *i* has called subcomponent *j*, or subcomponent *i* has returned from subcomponent *j*. Hence, if for example *find* is called from *AccountControllerBean*, Monitor executes a state transition and updates the current state vector.

The FSM for the application is obtained by observing the calls and returns between components from traces collected when the application is exercised with a given workload. We define a *profile* as a set of operations that a web user performs in the application such as listing account histories, transferring funds between accounts, or withdrawing funds. When imposing the workload to the Duke's Bank application for generating the FSM, we try to be as exhaustive as possible in the profiles, so that all the possible web interactions are executed. This is a critical step when generating the FSM, because failing to execute some of the web interactions can render an incomplete FSM.

Our FSM is composed of a total of 31 states and 62 events (two times 31 because of the calls and returns). The initial and final states of the FSM are the same and represent the state when a web interaction request is received by the server and when it is replied to the client respectively.

2.4 Rule Types and Syntax

In our previous work [5] we developed syntax for rule specification that can be applied in message-based applications. We now have extended the rule syntax to be more flexible so that they 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 following are examples by which response time can be derived to come up with temporal rules in Monitor: (1) a protocol specification may specify that an entity *i* should acknowledge or reply a request made from entity *j* in a bounded interval of time, (2) a Service Level Agreement (SLA) may specify QoS constraints for Web service components when delivered to specific users [13]—since subcomponents can be seen as service providers to other subcomponents, Monitor's temporal rules can be used to verify their expected response time, (3) models based on performance analysis tools, such as Magpie, can be used to derive normal response time of elements in three-tier applications such as e-commerce and online banking applications.

```
1 Type1  (/atmAck.jsp,service) 0 0
         (/atmAck.jsp,service) 0 100
2 Type2  (TxControllerBean,transferFunds) 0 75
3 Type2  (/accountList.jsp,service) 0 50
...
...
```

Fig. 2.2. Example of rules that are given to Monitor for verifying Duke's Bank Application behavior

Our rule syntax is able to specify response time lower and upper bounds at the granularity of function calls within a component. We explain two types of rules that can be applied directly to RPC-style applications—other types of rules that Monitor is able to handle, for example, rules for message-based applications, can be found in [5]:

1. **Type 1**: (S_i) for $T \in (t_N, t_N + k) \Rightarrow (S_j)$ for $T \in (t_M, t_M + l)$, where $t_M > t_N$, and $k, l \geq 0$.

   Explanation: This rule represents the fact that, if for some time interval $T \leq k$ starting at t_N, application is in state S_i, it should move to state S_j for some time interval $T \leq l$ starting at time t_M.

   Purpose: This rule serves to verify the time elapsed in a nested call of different subcomponents. Line 1 in Fig. 2.2 shows an example of this rule. If subcomponent

(*/atmAck.jsp*, *service*) calls another subcomponent which then calls another one—generating a sequence of calls—this example shows that we expect that a return should go to (*/atmAck.jsp*, *service*) in no more than 100 milliseconds. We can use this rule to model the maximum amount of time that a web interaction can take if state 0 is used in S_i and S_j.

2. **Type 2:** (S_i) for $T \in (t_N, t_N + k)$.

   Explanation: This rule represents the fact that given the rule is instantiated at time t=0, the application must be in state S_i for some time within t_N and t_N+k. This gives that the minimum and maximum time the application must stay in the state are t_N and t_N+k.

   Purpose: this rule serves to verify the minimum and maximum allowable response time of a subcomponent. Second and Third line in Fig. 2.2 show examples of these rules. In Line 2 we verify that (*TxcontrollerBean*, *transferFunds*) subcomponent responds in no more than 75 milliseconds, whereas in line 3 we say that (*/accountList*.jsp, *service*) should respond in no more than 50 milliseconds.

3. HANDLING HIGH STREAMING RATES: INTELLIGENT SAMPLING

3.1 Sampling in Monitor

With an increase in the incoming message rate at Monitor, the latency of detection can increase. In order to maintain an acceptable given latency, Monitor cannot afford processing all the messages coming at a high rate—processing all the messages has a high cost in terms of computation. To avoid being overwhelmed with the high computational cost, Monitor chooses to process only a fraction of the incoming messages. In our previous work [9], we present a scheme for randomly sampling a message out of *m* messages in order to reduce the computational workload in Monitor.

Sampling a message means that the message is consumed in Monitor for further rule matching. 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 drop the message or not. Therefore, if a message is sampled, it will be consumed by the StateMaintainer to prune the state vector, and consumed by the RuleMatching engine for rule matching.

When a message is dropped, Monitor cannot perform state transitions in the FSM, losing track of the actual state of the application. In this scenario, Monitor opts to maintain a set of possible states that the application could have reached. The fact that Monitor does not know the correct application's state is called *state non-determinism*. When sampling, depending on the number of consecutive dropped messages, the state vector can grow to a maximum of the total possible states in the FSM. As an example, consider a part of an FSM in Fig. 3.1. Suppose that the application is in state *A* at time t_1, and that a message is dropped. From the FSM, Monitor determines that the application can be in state *B* or state *C*, so the state vector $\omega = \{ S_B, S_C \}$. If another message is dropped at time t_2, the state vector grows to $\omega = \{ S_B, S_D, S_E, S_F \}$.

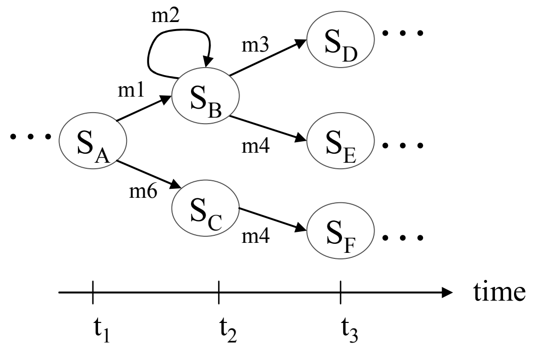


Fig. 3.1. A portion of a Finite State Machine.

Monitor's RuleMatching engine matches rules for all the states in ω once it is passed by the StateMantainer engine. 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 in the StateMaintainer and RuleMatching engines leading to increased detection latency. Even worse, a large and inaccurate state vector degrades the quality of detection through increase false alarms and missed alarms. A challenge is then to keep the state vector size bounded so that Monitor performs detection only corresponding to the correct application state.

3.2 Intelligent Sampling Approach

We hypothesize that sampling based on some inherent property from the FSM can lead to reducing the state vector size when pruning. We have observed that messages in the application have different properties in the FSM. For example, some messages can appear in different transitions while other appears in only one. Suppose for example that state vector ω = { S_B, S_C } following Fig. 3.1. 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 }. As we can see, the fact that m_3 appears in one transition while m_4 appears in two ones makes a difference in the resulting state vector. We say therefore that m_3 has a more *desired* property than m_4 in terms of sampling.

We propose an *intelligent sampling* approach whereby the incoming packets are *observed*, and a message with a desirable property is sampled. A packet is observed by determining the type of message it represents in the application. In network protocols, this can be performed by observing the header and payload of the packet, while in RCP-style applications this can be performed by observing the payload only. Let us denote by d_m, discriminative size, the number of times a message m appears as a state transition in the FSM. We say that in our intelligent sampling approach, a message with a small d_m is desired.

The procedure of selecting a message with a desirable property in a window of m messages is implemented in Monitor though a greedy algorithm. The details of the algorithm are explained in the next subsection.

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. 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 advanced what the next message would be, Monitor could end up with no sampled messages at all by the end of the window if no messages with $d_m < d_{th}$ arrive. 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*.

Fig. 3.2 shows the algorithm `ChooseMessageIntelligently` for intelligent sampling. Since this algorithm runs for a window of m messages, a main while loop in lines 5-16 is maintained to guarantee that only m messages are examined for every run, while the condition in line 6 guarantees that only n messages are sampled. If the last resort point is never reached (e.g., the desired messages arrive early in the window), the algorithm runs always lines 7-11. Here discriminative size of messages is examined to sample the desired ones. Otherwise, as a result of reaching the last resort point, it runs lines 12-14. Notice that in lines 7-11, each message is looked in a pre-computed static table which has the d_m for all the messages in the FSM. If a message is seen to have a $d_m < d_{th}$, it is sampled.

`ChooseMessageIntelligently` 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 discriminating 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

```
ChooseMessageIntelligently(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 then
13.      SampleMessage (currentMsg)
14.        numSampled ← numSampled + 1
15.  currentMsg ← getNextMessage( )
16.  numMsgs ← numMsgs + 1
17. return
```

Fig. 3.2. Pseudo-code for sampling message in a window.

The function *getNextMessage*( ) captures the next message from the wire. The function *SampleMessage*( ) passes the message to the StateMaintainer for further pruning of the state vector.

For a given window of m messages, the computational cost of this algorithm is O($K \cdot m$), where K is the cost of looking for the discriminative size of a message in *table*. We implemented *table* by using a hash-table so the expected time of this search is O(1), allowing an overall complexity that tends to O(m).

4. REDUCING NON-DETERMINISM: HMM-BASED STATE VECTOR REDUCTION

There are two major disadvantages when pruning the state vector even with the intelligent sampling approach. First, when a message is sampled using intelligent sampling 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 if the FSM is completely connected (or close to it). 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. 3.1, 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 faulty component, for example, a component that makes an unexpected call to another component as a resort to mask an error.

In order to overcome these difficulties, we propose the use of a Hidden Markov Model to determine probabilistically the current state of the application.

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 or observation can be generated according to an associated probability distribution. In contrast to a regular Markov Model where states are directly visible to the observer, in an HMM, only the observations are visible; states are *hidden* to the outside, hence the name Hidden Markov Model.

The main challenge of Monitor when handling non-determinism is to determine the correct state of the application when only a partial set of the occurred events is observed, that is, a partial set of messages is sampled. This phenomenon can be modeled with an

HMM because, from the perspective of Monitor, the correct state of the application is hidden when one or more messages are dropped. Therefore, we propose the use of an HMM to determine the probability of the application being in each of a set of states. In a subsequent stage, states with probability values below a threshold will be pruned from the state vector.

An HMM can be characterized by the following [14]:

1. $S = \{s_1, s_2, \ldots, s_N\}$, the set of N states in the model. A particular state at time t is denoted by q_t.
2. $V = \{v_1, v_2, \ldots, v_M\}$, the vocabulary of M distinct observation symbols, where v_i, $i = \{1, 2, \ldots, M\}$ is an individual symbol. We denote an observation sequence $O = \{O_1, O_2, \ldots, O_T\}$, where each observation O_t is one of the symbols from V, and T is the number of observations in the sequence.
3. The state transition probability distribution $A = \{a_{ij}\}$, where
$$a_{ij} = P(q_{t+1} = s_j \mid q_t = s_i),$$
$$1 \leq i \leq N,\ 1 \leq j \leq N;\ t = 1, 2, \ldots$$
4. The observation probability distribution in state j, $B = \{b_j(k)\}$, where
$$b_j(k) = P(v_k \text{ at } t \mid q_t = s_j),$$
$$1 \leq j \leq N,\ 1 \leq k \leq M$$
5. The initial state probability distribution $\pi = \{\pi_i\}$, where
$$\pi_i = P(q_1 = s_i),\ 1 \leq i \leq N.$$

We use $\lambda = (A, B, \pi)$ as a compact notation for the HMM as proposed in [14].

4.2 HMM Model Parameter Estimation and Training

We used the Baum-Welch algorithm [14] to estimate the HMM parameters in order 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 users for about 5 minutes. These are the same set of traces used to build our application's FSM. In generating the load, we tried to be as exhaustive as possible to produce a complete list of all the web interactions that can occur in the application.

When training the HMM, the Baum-Welch algorithm is run with an initial estimate of HMM parameters, A, B and π. Initial state probability distribution is considered to be uniform, that is, if there are N states then the initial probability of each state is $1/N$. Uniform distribution is assumed for the transition probabilities for all the edges outgoing from any state. (Edges that do not appear as a transition in the model have an initial probability of zero.)

4.3 Algorithm for Reducing the State Vector using HMM

We have implemented the `ReduceStateVector` algorithm for reducing the state vector with the HMM. When sampling a message, and before pruning the state vector, Monitor asks to the HMM for the *k* most probable application's states. Monitor then updates the current state vector by taking the states in common with the HMM's answer. Then the state vector is finally pruned according to the sampled message. Fig. 4.1 shows the pseudo-code for the algorithm which we now explain in detail.

`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's 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: new state vector 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 (the states in the model)

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. 4.1. Pseudo-code for the reduction of the state vector with the HMM.

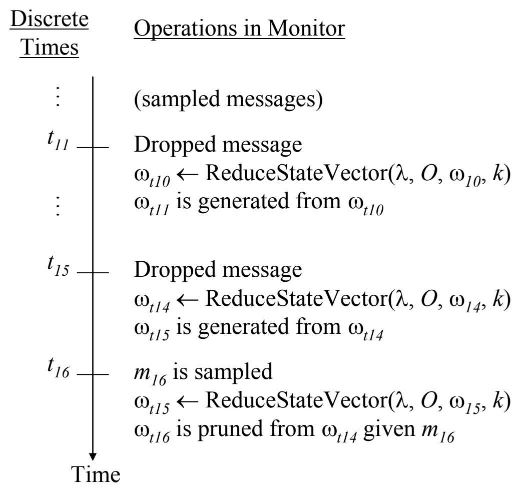

Discrete Times
Operations in Monitor
(sampled messages)
t_{11}
Dropped message
$\omega_{t10} \leftarrow$ ReduceStateVector(λ, O, ω_{10}, k)
ω_{t11} is generated from ω_{t10}
t_{15}
Dropped message
$\omega_{t14} \leftarrow$ ReduceStateVector(λ, O, ω_{14}, k)
ω_{t15} is generated from ω_{t14}
t_{16}
m_{16} is sampled
$\omega_{t15} \leftarrow$ ReduceStateVector(λ, O, ω_{15}, k)
ω_{t16} is pruned from ω_{t14} given m_{16}
Time

Fig. 4.2. Points in time when the ReduceStateVector algorithm is invoked. Each discrete time represents the time when a messages arrives to Monitor.

The `ReduceStateVector` algorithm consists of three steps:

Step 1—Calculate the probabilities of the application being in a state s_i for all the possible states. Here we ask to the HMM: What is the probability that, after seeing a sequence of messages, the application is in state s_1, s_2,…, s_N? This is expressed as P($q_t = s_i \mid O, \lambda$), where O is the sequence of messages until a message is sampled, and λ is the HMM. The maximum size of the sequence O will be the maximum size of a web interaction in terms of the number of messages. This step produces a vector of probabilities μ_t and it is executed within lines 1–3 in the pseudo-code.

Step 2—Sort the vector μ_t by the probability value of each element. This is done because we want to determine the most likely state of the application from the HMM answer, and because μ_t contains the probabilities for all the states. This step produces a new vector of probabilities α_t and it is executed in line 4.

Step 3—Compute an updated state vector ω_{t+1} by calculating the intersection of the current state vector ω_t and the top n elements in α_t. By using a small n, Monitor is able to reduce the state vector to few states. For example, taking the example in Fig. 3.1, 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 n=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 mechanism to reduce the non-determinism produced by pruning the state vector deterministically with an incorrect message. Having the intersection of ω_t and α_t equal to null implies that either the HMM or ω_t is incorrect. Therefore taking the union is a safe way to go. This step is executed within lines 5-11.

The time complexity of the algorithm is proportional to the time in computing μ_t, that is, the probabilities 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 n elements in α_t. The vector μ_t can be computed in time $O(N^3 \cdot T)$, 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 $\alpha_t[1 \ldots n]$ can be performed in $O(N \cdot n)$. Hence, the overall time complexity is $O(N^3 \cdot T + N \log N + N \cdot n)$. In practice, the last factor, $N \cdot n$, tends to be N because we select a small value for n, such as 1 or 2.

A disadvantage of this algorithm is that an application represented with an FMS composed of a large number of states can make expensive the use of the algorithm. However, for the Duke's Bank application we have an FSM of 62 states which makes this algorithm computationally feasible. Even when our HMM takes as input complete sequences of messages, that is, no messages are dropped for the HMM, the computational cost of this is less expensive than sampling all the messages, that is, processing them all through the StateMaintainer and RuleMatching engine.

The schematic in Fig. 4.2 presents points in time when the algorithm is invoked in the StateMaintainer. Notice that when a state vector is generated according to the figure it represents that it has been updated given the states in the previous state vector and that a message has been dropped.

5. EXPERIMENTAL TESTBED

We implement our new algorithms—intelligent sampling and HMM-based state vector reduction—in the context of the Monitor detection system. We test Monitor's performance for detecting errors in the Duke's Bank application, and compare it to two state-of-the-art detection schemes: the Convolution algorithm and Pinpoint. In this section we explain the main implementation details of Convolution and Pinpoint, and the tools that are used to test the three systems in a real operating scenario.

5.1 J2EE Application

We use the Duke's Bank Application as the testbed for injecting and detecting errors in a distributed environment. Duke's Bank provides administrative functionalities such as managing customers and accounts, and user functionalities such as accessing account's information and performing transactions. This application is representative of a medium-sized component-based application since it is composed of 4 web components (servlets and Java Server Pages) and 6 Enterprise Java Beans (EJB) components. The Duke's Bank application is presented in the J2EE Tutorial [11] and it is meant to demonstrate the functionality of the typical types of components encountered in e-commerce applications such as web components, EJB components and application clients.

5.2 Tracing in the J2EE System

Duke's Bank is run on Glassfish v2 [15], the open-source application server from Sun Microsystems. Glassfish provides the capability of collecting runtime information for each web interaction through the *CallFlow* monitoring system. The *CallFlow* package permits the gathering of runtime information for each web interaction as it flows through the containers in the application server. We use the *AsyncHandlerProducer* class in *CallFlow* to obtain the following runtime information: Request-ID (a key that identifies each web interaction), the caller and called component, the caller and called method, and a timestamp for each web interaction. This mechanism requires no application change and is less intrusive with respect to runtime overhead.

We use Request-ID to separate concurrent requests in the application. For each request that arrives to the web server, we create a web interaction that is composed of messages of the form <(*Request-ID*), (*timestamp*), (caller *component*, *method*), (called *component*, *method*)>. This level of granularity permits Monitor to detect problems not only at the level of web interactions, but also at the level of subcomponents.

5.3 Web-users Emulator

In order to evaluate our solutions in diverse scenarios such as high user requests rates and multiple types of errors, we wrote *WebStressor*, a trace-based web-users emulator. WebStressor emulates web interactions that are produced when the application is experiencing different user loads. WebStressor takes different profile's traces (obtained when building the FSM), and replays them by sending each message in the trace to the tested detection systems—Monitor, Convolution and Pinpoint—through datagram packets. Each profile's trace contains a sequence of web interactions that would be seen in *CallFlow* when a user of Duke's Bank application is executing multiple operations. We use a parameterized uniform random delay between web interactions to emulate the user think-time.

WebStressor can mimic *n* concurrent users performing operations on the application. Its configuration parameters are: the number of concurrent users, the think-time and the ramp-up delay (the time between two successive users start to interact with the system). For all the experiments presented in the next section, we wanted to create a specific number of users at a fast rate and impose a high stressing load on the detection 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 runs.

WebStressor also has error injection capabilities. It can mimic the injection of different kinds of errors in subcomponents in web interactions. These kinds of errors will be explained in Section 6.4.

5.4 Convolution Algorithm Implementation

The implementation of the Convolution algorithm (as presented in [1]) is still in progress and it is being done by another member of our research group; however, we use it as a point of comparison for Monitor in detecting performance failures. The convolution algorithm models delays at each component in the system using signal processing techniques. We devise a detection scheme from this algorithm and are implementing it in an online detection system to compare it with Monitor. We call this implementation Convolution in the rest of the paper. More details of the algorithm can be found in [1].

5.5 Pinpoint Implementation

Pinpoint [4] constitutes another point of comparison for Monitor in detecting anomalies in web interactions. Pinpoint proposes an approach for tracing paths from user requests and the use of a Probabilistic Context Free Grammar (PCFG) to detect failures in distributed applications. A PCFG is used to model normal path behavior and to detect anomalies whenever a path's structure does not fit the PCFG. (In contrast to Convolution, the implementation of Pinpoint has been completed and has been done by another member of the research group.)

A PCFG is a model typically used in natural language processing. It models the likelihood of a sentence based on a set of sentences used for training. In the context of our application, a sentence can be viewed as the method invocations and returns in a web interaction. A PCFG consists of a context free grammar (CFG) represented in Chomsky Normal form in which each production is assigned a probability after a training phase. As proposed in [4], we implement Pinpoint by using a PCFG to detect anomalies in web interactions. We call this implementation Pinpoint-PCFG in the rest of the paper.

We first generate a CFG in Chomsky-Normal Form automatically from the FSM. To explain how we build the CFG, consider a portion of a sample FSM in Fig. 5.1. For each edge in this FSM, we add two productions of the form $S_B \rightarrow SD_B\ S_C$ and $SD_B \rightarrow m1$, where S_B, S_C and SD_B are non-terminals and $m1$ is terminal in the grammar. For the return edge $m4$, we add an additional production $S_B \rightarrow m4$. All productions in CFG are of the form $NT \rightarrow NT\ NT$ and $NT \rightarrow T$, where NT denotes a non-terminal and T denotes a terminal.

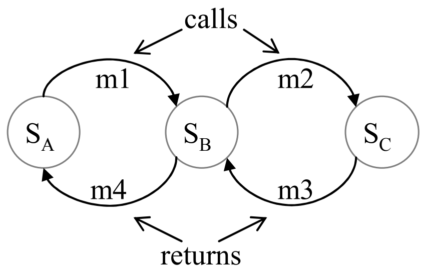


Fig. 5.1. A portion of a sample FSM representing the sequence of calls and returns for a particular web interaction.

Pinpoint-PCFG has a training phase and a testing phase (the online detection phase). Pinpoint-PCFG is trained using the same traces taken from Duke's Bank to build the FSM and to train the HMM in Monitor. We use the implementation of Inside-Outside Algorithm (IO) from Brown University [16] for the estimation of the production probabilities to generate a stochastic context free grammar. The IO Algorithm requires a context-free grammar and training sequence to determine the production probabilities.

The online part for the detection of failures by Pinpoint-PCFG is implemented by a statistical parser. We use Cocke-Younger-Kasami (CYK) parser algorithm to parse the sequence of messages received by Pinpoint-PCFG in a web interaction. We use the probabilistic CYK parser from Brown University [16] 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 given by the CYK parser be p_i. We calculate a measure M_j for the web interaction j by the following equation:

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

Note that, the higher the value of M_j for a web interaction seen, the lower is the probability of seeing that web interaction according to the PCFG model.

Pinpoint-PCFG's detection algorithm is a threshold-based algorithm. We compute M 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, M_{th} is an adjustable parameter.

One advantage of a PCFG, as pointed out in [4], is that it can detect some patterns that were not seen in the training phase. Since the grammar is context-free, the PCFG represents a super-set of the observed web interactions seen in the traces used for training.

6. EXPERIMENTS AND RESULTS

In this section we report experiments and results for comparing the performance of Monitor, Convolution and Pinpoint-PCFG in detecting errors in the Duke's Bank application. Our two new techniques (intelligent sampling and HMM-based state vector reduction) have been integrated in Monitor for the experiments.

6.1 Adjusting the Sampling Rate

When experiencing high incoming rates, Monitor starts sampling messages in order to avoid being overwhelmed. This is because after a certain incoming rate the latency in matching rules increases linearly from milliseconds to minutes and we would like the Monitor to operate at a rate less than where this linear increase starts. We call this point Monitor's *threshold rate* and denote it as R_{th}. We observe this threshold in Monitor when WebStressor emulates about 3 concurrent users in Duke's Bank application. Therefore, for the rest of this discussion we select the threshold as 3 concurrent users. We measured the average of incoming messages in Monitor for this threshold to be 52 messages/sec so we assign this value to R_{th}.

Sampling rate R_s in Monitor is adjusted by the formula:

$$R_s = 1 - \frac{(R - R_{th})}{R}, \qquad R > R_{th},$$

where R is the incoming rate.

The performance of the intelligent sampling algorithm is directly affected by the calculated sampling rate. The intelligent sampling 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*. Depending on the value of m used in Monitor, we can have a discretization error > 0. For example, if m = 8, and R_s = 0.741, the best factor Monitor can select is 6/8 = 0.75 which has a discretization error of 0.009.

Table 6.1
Sampling Factors calculated according to the number of messages per sec. received.

	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

The selection of *m* in the sampling factor is a critical step since it affects the performance of intelligent sampling algorithm. If *m* is greater than the size of a web interaction, intelligent sampling algorithm can drop all the messages in the web interaction. This can increase missed alarms because a fault injected in this web interaction will not be detected in Monitor. Therefore in terms of the rate of missed alarms, the ideal value for *m* should be the size of the smallest web interaction. We observe in the Duke's Bank application that the smallest size is 6, however we consider this a very small size for *m* because the discretization error when translating R_s to the sampling factor can be considerable. In order to avoid assigning a value to *m* much larger than 6 and that help us in managing a reasonable discretization error, we assign 8 to *m* for the rest of the experiments. Table 6.1 shows values for the sampling according to the incoming message rate, the best sampling factor and its corresponding discretization error. For the rest of the experiments, for each number of concurrent users shown in Table 6.1, we fix the corresponding sampling factor in Monitor.

6.2 State Vector Reduction

We now run experiments to confirm pragmatically our hypothesis that intelligent sampling algorithm 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 obligates Monitor to use a sampling rate of 50%, which we believe it is reasonable for assessing the sampling algorithms capabilities. This is run two times; one with Monitor configured in random sampling (RS) mode, and one configured in intelligent sampling (IS) mode. We then observe ω in the StateMaintainer engine in Monitor. The vector ω changes when a message is sampled as well as when a message is dropped. 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 for possible detection.

In each run, we obtained abut 3337 values of ω's size. Fig. 6.1 shows 100-samples snapshots of these values for RS and IS modes (the size of ω is shown for every message here.)

The high-peaks pattern that we observe in RS mode in Fig. 6.1 is due to the deficiency of random sampling in selecting messages with small discriminative size. Recall that, these messages have the desired property that they appear in one (or few) transitions in the FSM. In contrast we do not observe this pattern in IS mode, because it takes advantage of sampling most of these 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 bound to around 14. Also, ω's size can increase even when sampling is being done, but this happens less often with intelligent sampling due to its ability to sample suitable messages.

We observe in Fig. 6.1 that, for IS mode, ω's size is sometimes kept 1 consecutively for 4 times. We can observe the occurrence of this pattern one time in the range of samples 100−120, and two times in the range of samples 120−140 in Fig. 6.1. The reason for this is that 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 as much as it can, it is very likely that we observe such a pattern in ω. IS mode helps then in maintaining a ω's size of 1; our ultimate goal—regardless of the sampling algorithm, ideally ω should be of size 1 since the application can only be in one state. For RS on the other hand, the troughs are short-lived.

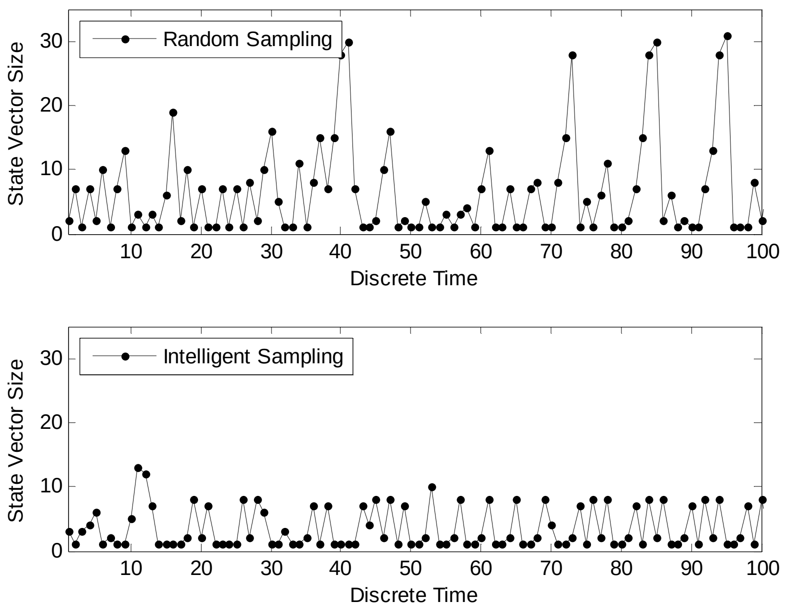


Fig. 6.1. Snapshot of 100 sampled values of the state vector in random sampling (RS) mode and intelligent sampling mode (IS) mode

We now measure only ω's size after it is pruned. Recall that the pruned state vector ω is the one used for rule instantiation and matching in the RuleMatching engine. Therefore, it is at this point in time that we more desire a reduction of ω for Monitor's improvement in detection accuracy and precision. Fig. 6.3 shows a cumulative distribution function (CDF) for the observed values of ω's size. As we expected, in IS mode, ω's size of 1 has a higher frequency of occurrence (about 83%) than in RS mode (60%). Thus, in IS mode, ω contains only one state most of the time. In contrast, all ω's size values > 1 have higher frequency of occurrence in RS than in IS. We observe as well that ω, after being pruned, can have a maximum size of 7. This is because of the nature of Duke's Bank application in which the maximum message's discriminative size is 7.

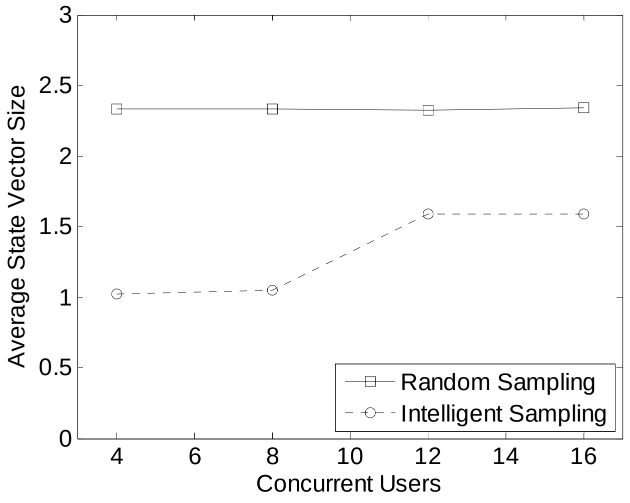


Fig. 6.2. Average size of the state vector over the run of an experiment

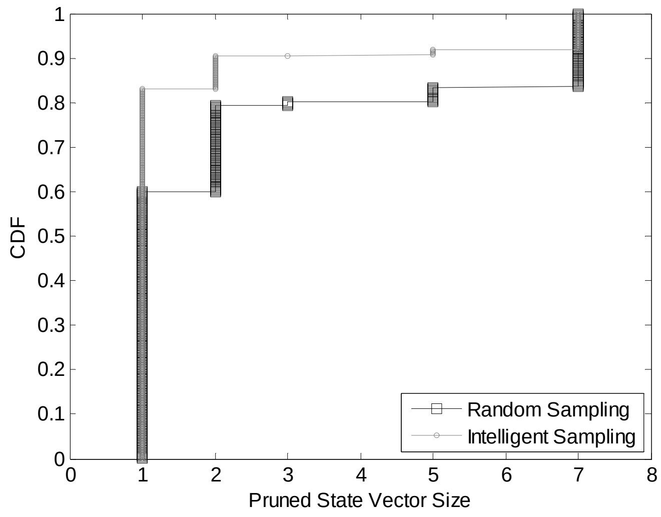


Fig. 6.3. Cumulative Distribution Function (CDF) for the pruned state vector with RS and IS

We vary the concurrent number of users in WebStressor order to observe the average ω's size under various loads imposed to Monitor. Fig. 6.2 presents the ω's average size for different numbers of concurrent users. Notice that, in average, the ω's size does not change considerably by varying the load charged to Monitor, neither for RS nor for IS mode.

One interesting observation is that the difference in ω's average size for RS and IS mode is small (in the order of 2 states) if we consider the total amount of states in our FSM, which is 31. Nevertheless, the cost of having a ω's size > 1 can be high in terms of computation and false alarms. First, if Monitor is fed with a base of complex rules, the tasks to be performed for matching them can be computationally intensive. For example, a rule can validate the consistency of data through different tables in a database. If an entry in a table is deleted, a consistency check has to be performed that all associated entries have been garbage collected, which can consume significant resources. Because the number of rules to be matched in Monitor is directly proportional to the number of states in ω, even a small reduction in ω's size can significantly reduce the computational cost in Monitor's RuleMatching engine. Second, and perhaps more importantly, a reduction of ω's size can reduce substantially the rate of false alarms—if we consider the worst case whereby all the *N* rules in Monitor are matched for any state, a reduction of *N* false alarms can be achieved, if ω's size is reduced by 1.

6.3 Definition of Performance Metrics

We introduce the metrics that we use to evaluate the Monitor's performance in detecting failures in web interactions. We use these metrics as well to compare the performance of Convolution and Pinpoint-PCFG to the one observed in Monitor.

Let *W* denote the set of web interactions generated in the application when a load of concurrent users is imposed. Recall that we are able to identify web interactions with a key as they start and finish in the application server. For every run of an experiment, we collect *W* and 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 actual correct detections made by Monitor).

We define *Accuracy* as $|C| / |I|$ and *Precision* as $|C| / |D|$, where $|C|$, $|I|$ and $|D|$ denote the number of web interactions in *C*, *I* and *D* respectively. In one hand, accuracy expresses how well the detection system is able to identify the web interactions in which problems occurred, while in the other hand, precision is a measure of the inverse of false alarms in the system.

An important performance metric of a detection system is the latency of detecting failures. Let T_i (time of injection) denote the time when a fault is injected. Also let T_d (time of detection) denote the time when the failure caused by a fault injected at T_i is detected in the detection system. We define *detection latency* as $T_d - T_i$. In the case when a delay δ is injected—emulating a performance problem in a component of the application—δ is subtracted from the total time, that is, detection latency = $T_d - T_i - \delta$. The reason δ is subtracted is that δ represents only a characteristic of the injected fault and does not reflect the level of performance 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 to be called. This can help in preventing the propagation of error to subsequent subcomponents and in diagnosing the root cause of the problem. In each experiment, we measure the time when a web interaction finishes and denote it as T_s. If the interval of time $T_d - T_s$ is < 0, we say that the detection system had a *pre-detection latency* of $| T_d - T_s |$, whereas if the interval of time $T_d - T_s$ is ≥ 0, we say that the detection system had *post-detection latency* of $T_d - T_s$. In Convolution and Pinpoint-PCFG, a complete web interaction has to be observed to perform error detection, while in Monitor detection is performed without looking at the entire web interaction. Therefore, Convolution and Pinpoint-PCFG always have post-detection, while Monitor may have pre- or post-detection.

6.4 Error Injection Model

Errors are injected into the application traces to test the performance of the three compared detection systems. Faults are injected by WebStressor at runtime when mimicking concurrent users.

We inject four kinds of errors that occur in real operating scenarios:

1. *Response delay*: a delay *d* is selected randomly between 500 msec and 750 msec, and is injected in a particular subcomponent. This error simulates subcomponent's response delays due to performance problems.
2. *Null Calls:* a called subcomponent is never executed. This error makes the web interaction get cut short and the client receives a generic error report, e.g., HTTP 500 internal server error.
3. *Runtime Exception*: this simulates undeclared exceptions, and declared exceptions that are not masked by the application. As in null calls the web interaction gets cut short and the client receives an error report.
4. *Incorrect Message Sequences:* this simulates an error that occurs in the application and for which there is an exception handler. The exception handler either continues with the current web interaction as is, or it changes the structure of the web interaction (e.g., by calling a different subcomponent). We change the structure of a web interaction by replacing the calls and returns in *N* consecutive subcomponents. The value of *N* is selected randomly between 1 and 5.

Of these kinds of injections, Convolution can only detect response delay, while Pinpoint-PCFG can only detect null calls, runtime exceptions and incorrect message sequences. Therefore, we carry out two separate set of experiments to compare Monitor's performance to Convolution and to Pinpoint-PCFG. The results of these experiments are shown in the next two sub-sections.

6.5 Detecting Performance Problems: Injection of Response Delays

In this subsection, we explain our evaluation of Monitor and Convolution in detecting performance problems in web interactions. We inject delays to simulate performance problems in the set of 5 components listed in Table 6.2.

A category of errors that is one of the most 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 any subcomponent in Table 6.2 is touched in a web interaction.

Table 6.2
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

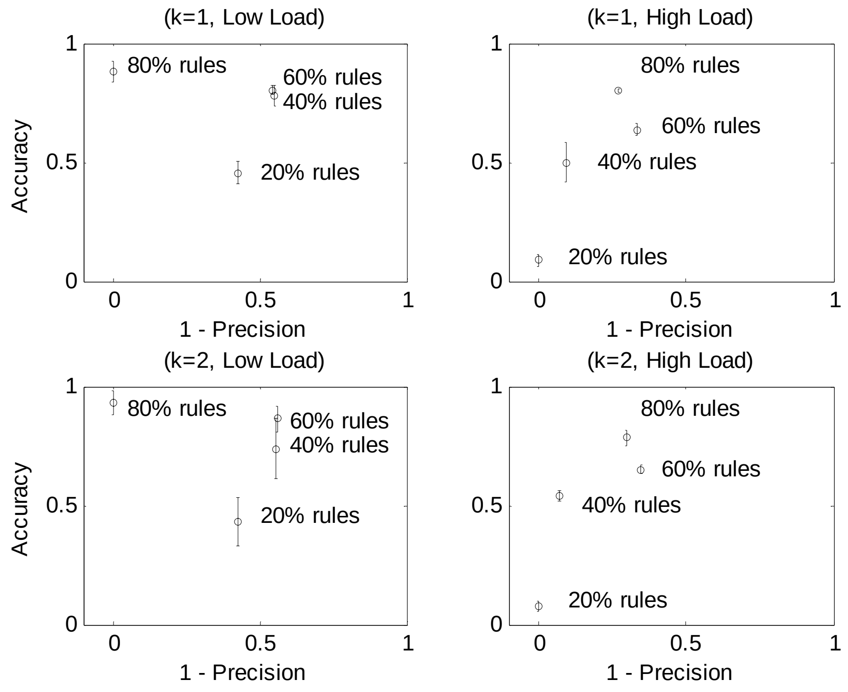


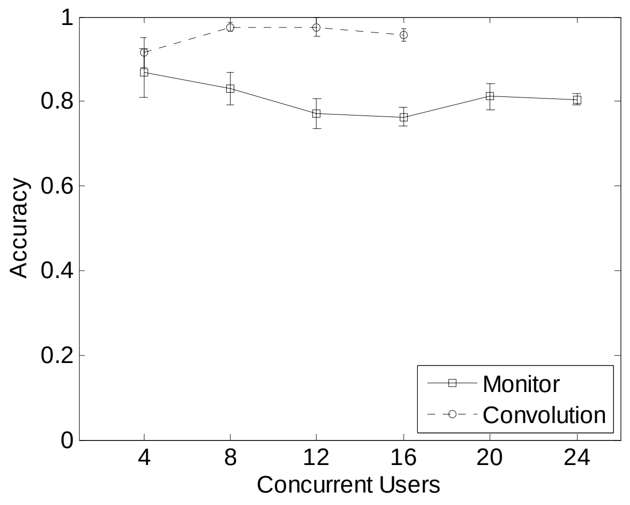
Fig. 6.4. ROC curves generated by varying the % of rules and parameter k in Monitor. Error bars show 95% of confidence intervals.

Before running the experiment, we determine the *best* operational parameters' set in Monitor and Convolution, that is, those that produce the best results in terms of accuracy and precision. For this we generate ROC (Receiver Operating Characteristic) curves for Monitor and Convolution by varying their configuration parameters and the imposed load of users to the application. We use two kinds of loads: *low load*, that is 4 concurrent users, and *high load*, that is 20 concurrent users. Once the ROC curves are generated, we select a point in the curves according to the following common criteria (we call this point as *operational point*):

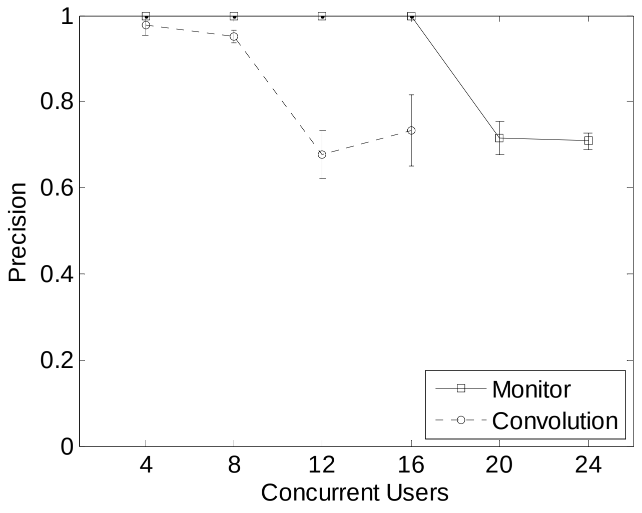
(a) Select the closest point to the *ideal point*, that is (0, 1). If more than one point fulfills this criterion, evaluate the next one.

(b) Select the point that has the best precision. If more than one point fulfills this criterion, the choice of the best operational point is left to the system administrator.

In Monitor, we vary two parameters: the size of the rulebase and k, an input to the HMM-based state vector reduction algorithm. Rulebase size is varied by activating randomly into Monitor only 20%, 40%, 60% or 80% of the exhaustive list of rules, while parameter k is varied over 1 and 2. The reason we do not vary k over values > 2 is that Monitor's HMM is not likely to help in detecting delays. Fig. 6.4 shows the ROC curves. We can 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. Also it is noticeable that the variation of k does not affect considerably the dispersion of the points in the two ROC curves. According to our criteria, we finally select as Monitor's best configuration parameters a rulebase size of 80%, and $k = 2$. This corresponds to the point closest to (0,1) in the first plot in Fig. 6.4.



(a)



(b)

Fig. 6.5. Accuracy and Precision for Monitor and Convolution. Error bars show 95% of confidence intervals.

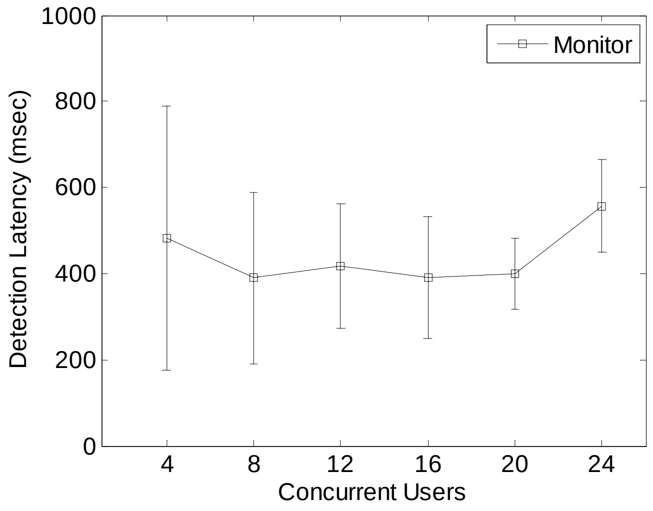


Fig. 6.6. Monitor's detection latency when detecting performance delays. Error bars show 95% of confidence intervals.

Delays injection experiment is run once operational points are selected for the two systems. Fig. 6.5 shows the results of this experiment. In Monitor, we observe a decrease in its accuracy of about 5% 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 of 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 traded-off. The decrease in precision is explained by how aggressive the rules are in Monitor. When the application is stressed with a high load, component's response time increases naturally, making Monitor to flag more false alarms. We can relax the rules by enlarging the amount of delay considered as normal for components, which would maintain a precision of 1 even when the application is exposed to higher loads. However, this can reduce accuracy since Monitor may not be able to detect a delay smaller than this bound.

Fig. 6.6 shows the results for the detection latency observed in Monitor. In average, detection latency is kept constant by Monitor. This is the direct effect of (1) adjusting the sampling rate in Monitor for different incoming messages rates and (2) the efficiency in the intelligent sampling algorithm in reducing the state vector size. Notice that if Monitor uses random sampling for this experiment, the same pattern would have been observed, but with a higher average. This is because the sampling rate in any sampling approach is selected so that it allows detection latency be adjusted to a required value.

6.6 Detecting Anomalous Web Interactions

We evaluate Monitor detection's performance in detecting anomalous web interactions by injecting null calls, runtime exceptions and incorrect message sequences. As in the delay injections, performance is evaluated by measuring accuracy, precision and detection latency. We also evaluate Pinpoint-PCFG's performance with the same categories of injections.

Monitor detects anomalous web interactions at the state maintainer by verifying the correct transitions in the FSM. If an event is unexpected according to the current state in Monitor's state vector, an error is flagged. This avoids the need for rules for this type of detections. All the detections by Monitor will point out the actual injections, as Monitor is verifying messages according to its FSM. Therefore we will have a precision of one at Monitor for this kind of injections. We show Fig. 6.7 rather than a ROC curve.

Under sampling, Monitor can drop an incorrect message, thus reducing accuracy. We show that in practice, our hypothesis that HMM is useful for detecting anomalous web interactions holds. Fig. 6.7 shows Monitor running with different values of *k* in the HMM algorithm. Parameter *k*=0 represents Monitor running without HMM. We can observe that, with no HMM, in both low and high load, 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).

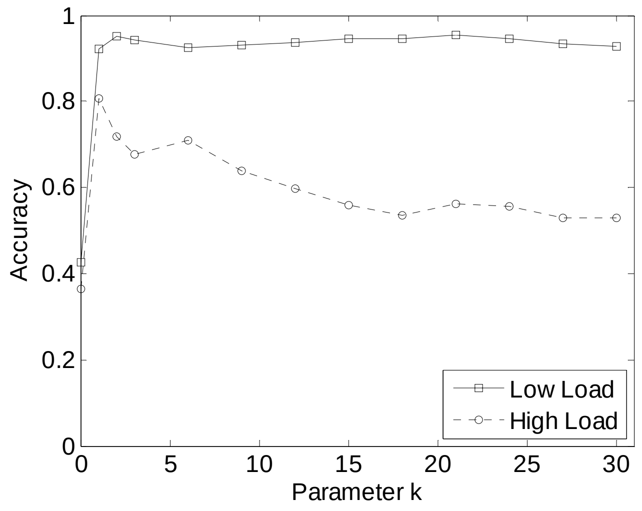


Fig. 6.7. Monitor's accuracy when varying parameter *k* in the HMM-based state vector reduction algorithm

In high load, two conditions cause Monitor to have a decreasing accuracy when increasing *k*: (1) The Monitor samples less often leading to an increase in the state vector size. With a large *k*, few states will get pruned and if the observed erroneous message is possible in any of the remaining states of the state vector, the error will not be detected. (2) Increasing *k* effectively reduces the impact of the HMM, since even states with low probabilities given by the HMM are being considered. Under high load, when the erroneous message may not be sampled, the HMM is particularly important. Therefore, under high load a high value of *k* is not a good option for Monitor. For the rest of the experiments, we use *k*=1 as it allows Monitor to have good accuracy in both low and high load.

We vary the threshold M_{th} (a parameter of Pinpoint-PCFG) to get Pinpoint-PCFG's ROC curves under low and high load. Fig. 6.8 shows the results of this experiment. A lower value of threshold generates more false positives as opposed to a higher one. A too high value would on the other hand generate missed alarms. We select M_{th} = 350 in the ROC curves as the operating point based on the same criteria as in section 6.5.

We observe that on average, Monitor's accuracy is comparable to Pinpoint-PCFG. In Monitor, accuracy decreases for higher load due to dropping more messages in a sampling widow. We see the robustness of Pinpoint-PCFG to false positives as it maintains on average almost the same precision with increasing number of users (0.9). As can be see in Fig. 6.9(b), precision line in Pinpoint-PCFG can shift up or down—this can vary in the PCFG based on the degree of training.

A trained PCFG represents a superset of the observed web interactions in our system. Therefore, PCFG can actually match some patterns that were not seen in the training phase. Still there will be some patterns that are normal but because they were not seen in the training phase, generate a false alarm.

We observe a mean accuracy value of 0.9 in Fig. 6.9(a) even with increasing number of users. As the load increases, Pinpoint-PCFG maintains a high accuracy because it is not dropping messages—messages are being enqueued for processing eventually. However the latency of detection suffers significantly at these high loads as seen in Fig. 6.10(b)—latency in Pinpoint-PCFG is in the order of seconds while in Monitor is in the order of milliseconds. Note that Monitor performs better on precision and maintains a close accuracy to Pinpoint-PCFG even with dropping of messages.

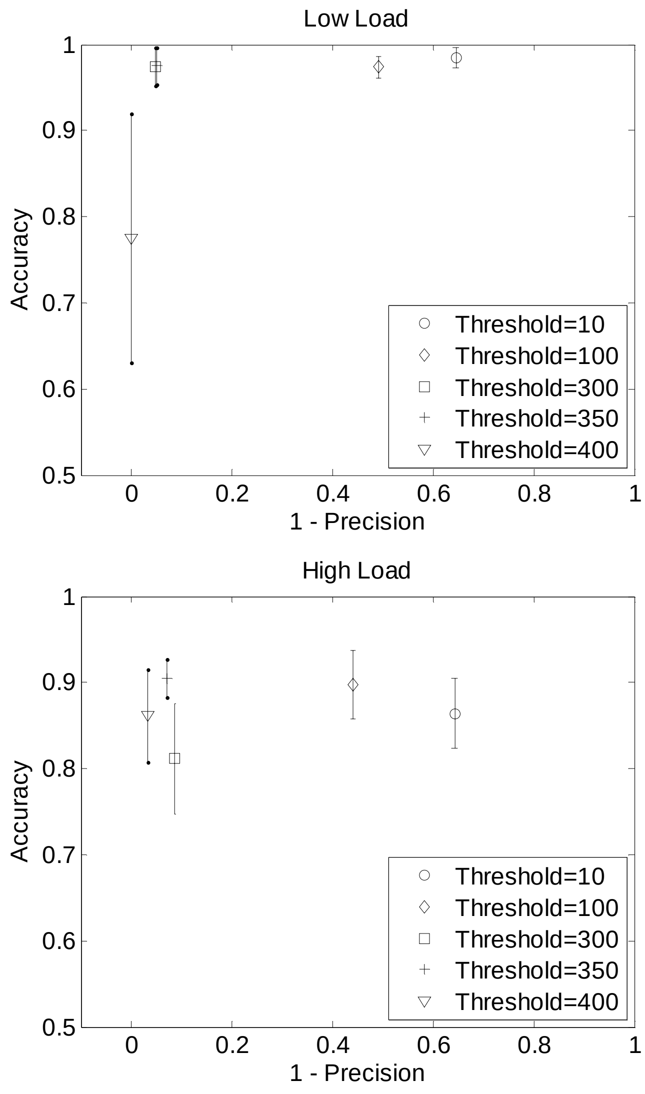


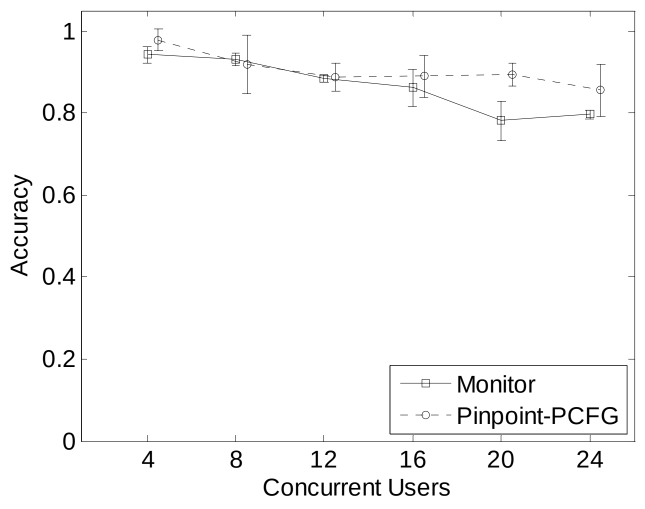
Fig. 6.8. ROC curve for PCFG. Points are generated by varying the Threshold M_{th}. Error bars show 95% of confidence intervals.

Detection latency in Monitor is dependent on how fast the state vector is pruned after sampling a message. The complexity of pruning is $O(N^2)$, where N is the number of states which is 31 in the FSM. Monitor is able to detect before a particular web interaction ends.

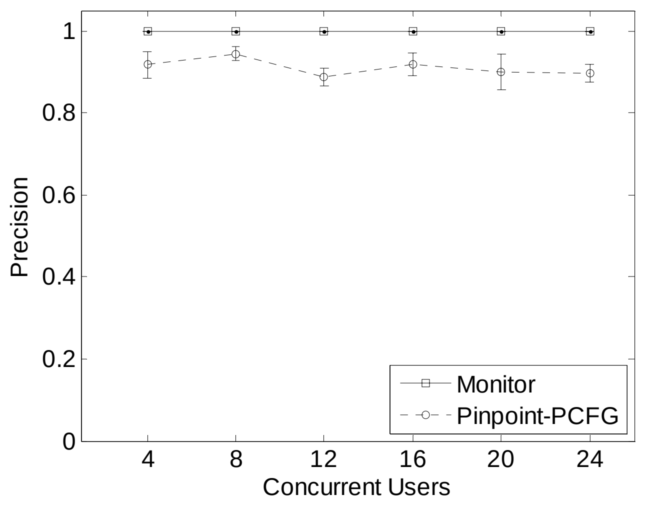
Table 6.3 shows the percentages of pre and post-detection delay for Monitor as we increase concurrent users. We observe that Monitor has a pre-detection latency that can vary from 8.93% to 22.67%.

The Pinpoint-PCFG's detection latency in the order of seconds can be explained by the time and space complexity of the parsing algorithm in the PCFG. Complexity of the recognition algorithm is $O(L^3)$ and space complexity is $O(RL^2)$, 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, and that the weighted average size is 70. Previous work with PCFGs [17] has also shown that, average time to parse sentences of length 40 can easily take 120 seconds even under optimized parameters scenarios.

Another cause of the high latency in our Pinpoint-PCFG implementation is the large amount of virtual memory that the process is taken. Due to its space complexity, Pinpoint-PCFG process consumes a large amount of virtual memory as can be seen in Table 6.4 in our memory consumption experiments (933.56 MB for a load of 24 concurrent users in the application). This makes Pinpoint-PCFG process to thrash in the machine where we conducted our experiments—a Linux box with 4 processors (3.40GHz) and 1024 MB of RAM.

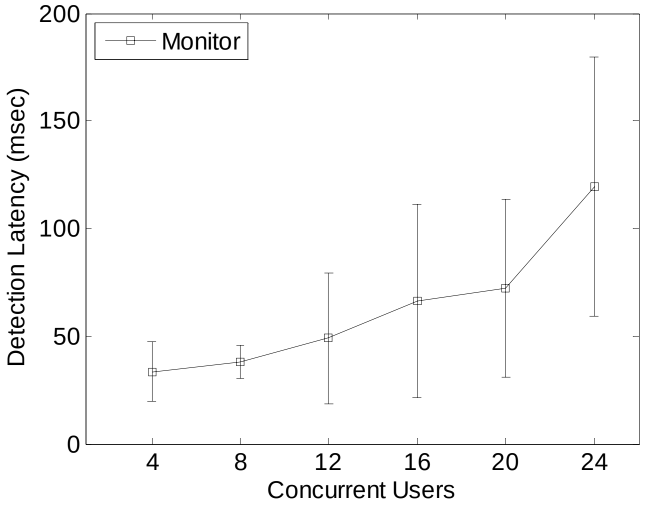


(a)

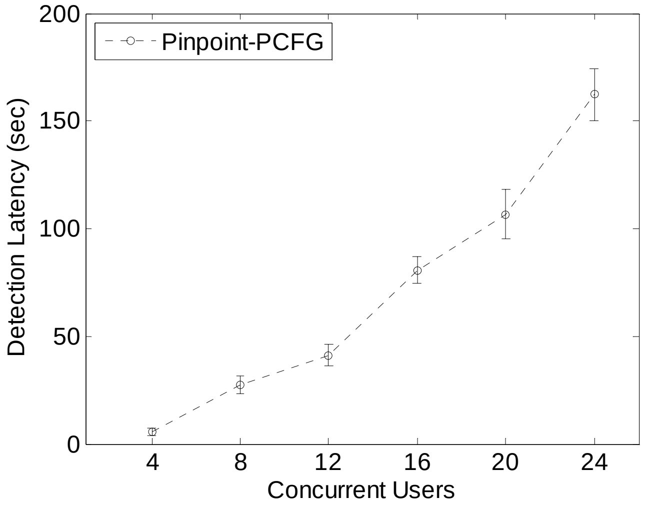


(b)

Fig. 6.9. Accuracy and Precision for Monitor and Pinpoint-PCFG. Error bars show 95% of confidence intervals.



(a)



(b)

Fig. 6.10. Detection latency for Monitor and Pinpoint-PCFG. Error bars show 95% of confidence intervals.

Table 6.3
Pre and Post detection delay for Monitor when detecting incorrect messages in web interactions

Type of Latency (%)	Concurrent Users					
	4	8	12	16	20	24
Pre-Detection	15.38	8.93	22.67	15.24	11.50	9.86
Post-Detection	84.62	91.07	77.33	84.76	88.50	90.14

6.7 Random and Intelligent Sampling Comparison

We evaluate the performance of random and sampling in detecting performance delays. Since intelligent sampling allows Monitor to reduce the state vector, it is expected the number of rules that are matched in is less than in random sampling, therefore affecting the accuracy and precision directly.

For this experiment, we use similar definitions for accuracy and precision as in the previous experiments, but we changed the granularity of detection from web interactions to subcomponents. Since the difference between intelligent and random sampling is reflected mainly in the number of instantiated and matched rules at runtime, we want a definition that permit us evaluate this difference fairly. For this experiment, let *W* denote the set of subcomponents where faults are injected in the application. Variables *I*, *D* and *C* are now defined as:

- *I*: out of *W*, the subcomponents where faults were injected,
- *D*: out of *W*, the subcomponents in which Monitor detected a failure,
- *C*: out of *I*, the subcomponents in which Monitor detected a failure

Accuracy and Precision formulas are the same than in subsection 6.3. Notice that this definition measure Monitor's performance in detecting errors at the subcomponents level. Detection at this level is helpful in diagnosis—finding the root cause of the problem—because it helps in reducing the set of subcomponents to look for the cause of the problem. Even when an error is pinpointed in a different subcomponent, a diagnosis scheme does not have to look for the problem in the set of all the subcomponents but in a reduced one.

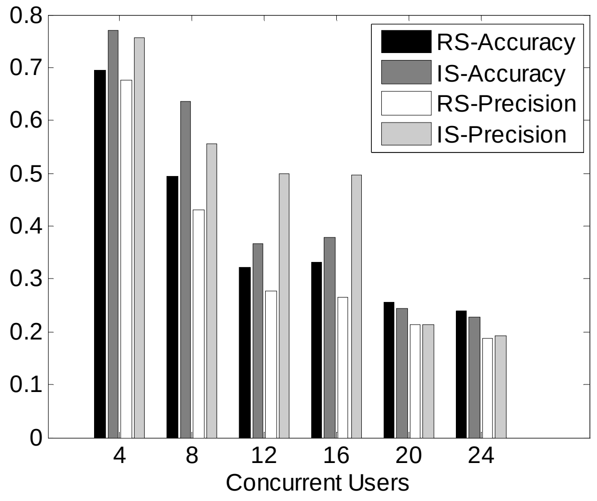


Fig. 6.11. Accuracy and Pecision for Random Sampling (RS) and Intelligent Sampling (IS).

We vary the number of concurrent users as in previous experiments, and measure accuracy and precision in Monitor running in random sampling and intelligent sampling separately. Delays are injected in the same subcomponents used in subsection 6.5 Fig. 6.11 shows the results of the experiment. We observe that accuracy and precision is higher for IS for almost all the application's loads; 4, 8, 12 and 16 concurrent users. However, for high loads, that is 20 and 24 concurrent users, random and intelligent sampling exhibit almost the same performance. The reason for this is that our definition does not allow us to measure whether detections in subcomponents in the set *C* are 100% correct. Due to the dropping nature inherent in both intelligent and random sampling, delays can be seen and detected in states where no error was injected. Since subcomponents are treated as states in our FSM, this causes high rates of false alarms at the subcomponent level. This leads to non-reliable accuracy measurements for high loads when this situation is highly exposed. If we were able to measure correct detections—elements in set *C*— at finer granular level, intelligent sampling would show better performance in the graphs even in high loads.

6.8 Memory Consumption

To see the highest amount of memory that our implementations were consuming, we measure average memory consumption for Monitor, Convolution and Pinpoint-PCFG under a load of 24 concurrent users. Memory consumption is collected every 5 seconds by reading the `/proc` filesystem in our Linux boxes (4 3.40GHz processors and 1024 MB of RAM) and averaged out the total duration of each experimental run. We measure the virtual memory size (VmSize) and the number of pages the process has in real memory (VmRSS) which we call memory in RAM.

The configuration parameters for each system are:

- Monitor: k=2 and 80% of the rules
- Pinpoint-PCFG: Threshold M_{th}=350
- Convolution: (implementation still in progress)

Table 6.4 shows the results of this experiment.

Table 6.4
Memory consumption for the three compared systems (Monitor, Pinpoint-PCFG and Convolution)

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

Monitor does not require storing large data structures at runtime that is the reason we see smaller values of average virtual memory and memory in RAM as compared to Convolution and Pinpoint-PCFG. However, Pinpoint-PCFG recognition algorithm's high space complexity O(RL^2) along with time complexity of $O(L^3)$ results in the high memory consumption as seen in Table 6.4. The Cocke-Younger-Kasami (CYK) algorithm, the parsing algorithm used in the implementation, requires storing and updating the probabilities in a three-dimensional array for each web-interaction encountered. Due to the large average sizes of web interactions (in terms of the number of messages) seeing at runtime, the virtual memory consumption on average is in the order of 933 MB. Since the implementation of Convolution is still in progress, we do not present its memory consumption results in this work.

7. RELATED WORK

7.1 Error Detection in Distributed Systems

Previous approaches to detection in distributed systems have varied from heartbeats to watchdogs [18][19][20]. 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 that is more related to ours is Pinpoint [4]. Authors in [4] present an approach for tracing paths from user requests and propose the use of statistical analysis techniques to detect and diagnose failures in large distributed systems. Particularly, a Probabilistic Context Free Grammar (PCFG) is used to model normal path behavior, that is, to model the likelihood of a given path occurring based on the paths 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. Authors claim that their techniques can be used to detect structural changes in paths as well as performance anomalies, for example, when a non-responsive or sluggish component stalls a path. Pinpoint does not consider the problem of dealing with high rates of requests. In contrast, in our work we propose sampling approaches to cope with high rates of incoming data. We believe that the detection system's performance should not suffer abruptly because of high loads—rates of missed and false alarms, and delay in detecting failures should not go below an acceptable level.

7.2 Performance Modeling and Debugging

There is an increase of work in providing tools for debugging problems in distributed applications—Project5 [1][21] and Magpie[2][3]. The general flavor of the approaches is that tools collect trace information at different levels of granularity which are used for automatic analysis, often offline, to determine the possible root causes of the problem.

For example, Project5's main goal is detecting performance delays on distributed systems. In [1] models for performance delays on RPC-style and message-based application for LAN environments are proposed—authors focus on finding high latency causal path patterns. Authors propose a *nesting algorithm* that examines a global system trace and establishes relationships of nested calls between components in the system. A second algorithm, *convolution algorithm,* separates the system trace as per function call trace, and models the delays at each component based on signal processing techniques. Convolution algorithm applies to both types of applications—RPC-style and message-based applications. In [21] authors present a third algorithm for performance debugging in wide-area systems. The algorithm, called *message linking algorithm,* models causal path structure and its timing in wide-area systems by taking into account network latency.

The Magpie project [2][3] 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. This is performed by the use of Event Tracing, a logging infrastructure built for the Windows operating system. It then constructs workload models that discard the scheduling artifacts due to OS multitasking, timesharing and caching. These models can be used for capacity planning, performance debugging and anomaly detection. We believe that Magpie is complimentary to our work, because workload models of request behavior could be used in Monitor to specify rules aimed in the detection of anomalies and performance bottlenecks.

Other powerful tools have also been proposed recently. For example, in [19] authors present a tool called *liblog* that aids in recreating the events that occurred prior to and during failure. The replay can be done offline at a different site. The tool guarantees that the event state in its log will be consistent, that is, no message is received before it has been sent.

7.3 Stateful Detection in High Throughput

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. For example, [22] proposes a partitioning approach of the network traffic to make the stateful analysis manageable by a NIDS in high-speed links. The idea in [22] is to divide the traffic volume in smaller portions for different devices (sensors), in such a way that guarantees the detection of all the attack scenarios. Another approach is increasing the efficiency of the detection pattern-matching algorithms (e.g., [23]). This research work differs from ours in two aspects. First, 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. In our work, we want to address the issue of making the detection task manageable first at the level of single machines, which will allow any detection infrastructure scale up. Second, they look for problems at the network level by the use of misuse rules. Our work looks at application level deviations from expected behavior.

7.4 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, [24] proposes 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 (or pair of addresses), and the value can be the number of bytes or packets. A sketch can indicate if any given key exhibits large changes, and can provide accurate probabilistic estimates of the changes. Authors in [24] build a forecast model for the changes in rates of different kinds of network packets. Sampling techniques has also been used in high-speed links as input for anomaly detection [25][26], for example, for detecting denial-of-service (DoS) attacks or worm scans, while some studies show that these sampling techniques introduce fundamental bias that depredates performance when detecting network anomalies (e.g., in [27]). Our work differs from those studies fundamentally because we focus on detection of failures at the application level rather than only at the network layer, and because we perform stateful detection by applying rules to states in the monitored application.

8. CONCLUSIONS

In this work, we have tackled the problem of state non-determinism in Monitor (our failure detection system) caused by sampling messages when performing stateful detection in distributed applications. An intelligent sampling algorithm has been incorporated in Monitor to select particular messages so that non-determinism is minimized, and its accuracy and precision in detection is increased as compared to randomly selecting the messages. In addition, an HMM-based technique has been incorporated in Monitor to determine the most likely application's state(s) so that rule matching is applied to only those states. We use the Duke's Bank application as a sample of a distributed application in which errors are injected and detected while imposing a load of concurrent users.

We have shown that the reduction of the state vector caused by the intelligent sampling approach has a direct effect in Monitor's performance; accuracy and precision in detecting performance problems in application's subcomponents, and in detecting anomalous web interaction, is comparable or superior to other state-of-the-art detection systems: Convolution and Pinpoint. Also, our HMM-based technique has shown to be effective in making Monitor robust in detecting faulty web interactions. For example, accuracy in Monitor when detecting anomalies in the structure of web interactions is similar to Pinpoint (about 0.8), while Monitor's precision is superior (it is 1.0 as compared to an average of 0.9 in Pinpoint).

In detection latency, Monitor outperforms the other two systems. For example, in detecting anomalies in web interactions, Monitor provides detection latency in the order of milliseconds, while Pinpoint performs detection in the order of seconds. This considerable difference is due to the fact that Monitor sample messages to reduce its workload while maintaining a high level of accuracy and precision; however, Pinpoint does not sample, thus it analyses every message in all the observed web interactions. We do not present results of detection latency for Convolution because its implementation is still in progress; nevertheless, we expect Convolution’s detection latency to be higher than Monitor because, as Pinpoint, Convolution does not sample messages.

LIST OF REFERENCES

LIST OF REFERENCES

[1] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds and A. Muthitacharoen, “Performance Debugging for Distributed Systems of Black Boxes,” *ACM Symposium on Operating Systems Principles* (SOSP ‘03), Vol. 37, Issue 5, Dec. 2003, pp. 74-89.

[2] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, “Magpie: Online Modeling and Performance-aware Systems,” *USENIX Hot Topics in Operating Systems* (HotOS ‘03), Vol. 9, May 2003, pp.15-15.

[3] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, “Using Magpie for Request Extraction and Workload Modeling,” *USENIX Symposium on Operating Systems Design & Implementation* (OSDI ‘04), Vol. 6, Dec. 2004, pp. 259-272.

[4] M. Y. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, “Path-based Failure and Evolution Management,” *USENIX Symposium on Networked Systems Design and Implementation* (NSDI ‘04), Mar. 2004, pp. 309–322.

[5] G. Khanna, P. Varadharajan, S. Bagchi, "Automated Online Monitoring of Distributed Applications through External Monitors," *IEEE Trans. on Dependable and Secure Computing*, Apr.-Jun. 2006, Vol. 3, No.2, pp.115-129.

[6] C. Kruegel, F. Valeur, G. Vigna and R. Kemmerer, "Stateful Intrusion Detection for High-Speed Networks," *IEEE Symposium on Security and Privacy*, 2002, pp. 285-293.

[7] Y. Wu; S. Bagchi, S. Garg and N. Singh, "SCIDIVE: A Stateful and Cross Protocol Intrusion Detection Architecture for Voice-over-IP Environments," *IEEE Int. Conference on Dependable Systems and Networks* (DSN ’04), Jun. 2004, pp. 433-442.

[8] H. Dreger, A. Feldmann, V. Paxson and R. Sommer, “Operational Experiences with High-Volume Network Intrusion Detection,” ACM Conference on Computer and Communications Security (CCS ’04), Oct. 2004, pp. 2-11.

[9] G. Khanna, I. Laguna, F. A. Arshad and S. Bagchi, "Stateful Detection in High Throughput Distributed Systems," *IEEE International Symposium on Reliable Distributed Systems* (SRDS ‘07), Oct. 2007, pp.275-287.

[10] C. Warrender, S. Forrest, B. Pearlmutter, "Detecting Intrusions Using System Calls: Alternative Data Models," *IEEE Symposium on Security and Privacy*, May 1999, pp.133-145.

[11] The Java EE 5 Tutorial (for Sun Java System Application Server 9.1). http://java.sun.com/javaee/5/docs/tutorial/doc/, Sep 2007

[12] N. Kothari, T. Millstein, R. Govindan, "Deriving State Machines from TinyOS Programs Using Symbolic Execution," *International Conference on Information Processing in Sensor Networks* (IPSN '08), Apr. 2008, pp.271-282.

[13] A. Dan et al, "Web Services on Demand: WSLA-driven automated management," *IBM Systems Journal*, Mar. 2004, Vol. 43, No. 1, pp.136-158.

[14] L.R. Rabiner, "A Tutorial on Hidden Markov Models and Selected Applications in Speech Recognition," *Proceedings of the IEEE*, Feb. 1989, Vol. 77, No.2, pp. 257-286.

[15] GlassFish—Open Source Application Server. https://glassfish.dev.java.net/, 2008.

[16] Open Source Software Written by Mark Johnson. http://www.cog.brown.edu/~mj/Software.htm, Mar 2008.

[17] D. Klein and C. D. Manning, "Parsing with treebank grammars: empirical bounds, theoretical models, and the structure of the Penn Treebank," *The Annual Meeting on Association For Computational Linguistics*, Jul. 2001, pp. 338–345

[18] H. J. Wang, J. Platt, Y. Chen, R. Zhang, and Y. M. Wang, "PeerPressure for automatic troubleshooting," *Joint International Conference on Measurement and Modeling of Computer Systems*, Jun. 2004, pp. 398-399.

[19] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay Debugging for Distributed Applications," *USENIX Annual Technical Conference*, May 2006, pp. 289-300.

[20] W. Chen, S. Toueg, and M. K. Aguilera, "On the Quality of Service of Failure Detectors," *IEEE International Conference on Dependable Systems and Networks* (DSN '00), Jun. 2000, pp. 191-201.

[21] P. Reynolds, J. L. Wiener, J. C. Mogul, M. K. Aguilera and A. Vahdat, "WAP5: black-box performance debugging for wide-area systems," *International Conference on World Wide Web* (WWW '06), May 2006, pp. 347–356.

[22] C. Kruegel, F. Valeur, G. Vigna and R. Kemmerer, "Stateful intrusion detection for high-speed networks," *IEEE Symposium on Security and Privacy*, 2002, pp. 285-293.

[23] 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.

[24] B. Krishnamurthy, S. Sen, Y. Zhang, and Y. Chen, "Sketch-based change detection: methods, evaluation, and applications," *ACM SIGCOMM Conference on Internet Measurement*, Oct. 2003, pp. 234-247.

[25] P. Barford, J. Kline, D. Plonka, and A. Ron, "A Signal Analysis of Network Traffic Anomalies," *ACM SIGCOMM Workshop on Internet Measurment* (IMW '02), Nov 2002, pp. 71-82.

[26] A. Lakhina, M. Crovella and C. Diot, "Mining Anomalies Using Traffic Feature Distributions," *ACM SIGCOMM Comput. Commun. Rev.*, vol. 35, issue 4, Oct 2005, pp. 217-228.

[27] J. Mai, C. Chuah, A. Sridharan, T. Ye and H. Zang, "Is Sampled Data Sufficient for Anomaly Detection?," *ACM SIGCOMM Conference on Internet Measurement*, Oct 2006, pp.165-176.