

Distributed Diagnosis of Failures in a Three Tier E-Commerce System

Gunjan Khanna, Ignacio Laguna, Fahad A. Arshad, Saurabh Bagchi
Dependable Computing Systems Lab (DCSL)
School of Electrical and Computer Engineering, Purdue University
Email: {gkhanna, ilaguna, faarshad, sbagchi}@purdue.edu

Abstract

For dependability outages in distributed internet infrastructures, it is often not enough to detect a failure, but it is also required to diagnose it, i.e., to identify its source. Complex applications deployed in multi-tier environments make diagnosis challenging because of fast error propagation, black-box applications, high diagnosis delay, the amount of states that can be maintained, and imperfect diagnostic tests. Here, we propose a probabilistic diagnosis model for arbitrary failures in components of a distributed application. The monitoring system (the Monitor) passively observes the message exchanges between the components and, at runtime, performs a probabilistic diagnosis of the component that was the root cause of a failure. We demonstrate the approach by applying it to the Pet Store J2EE application, and we compare it with Pinpoint by quantifying latency and accuracy in both systems. The Monitor outperforms Pinpoint by achieving comparably accurate diagnosis with higher precision in shorter time.

1. Introduction

The connected society of today has come to rely heavily on distributed computer infrastructure, be it an ATM network, or the distributed multi-tier applications behind e-commerce sites. The consequences of downtime of distributed systems may be catastrophic. They range from customer dissatisfaction to financial losses to loss of human lives [1][2]. There is an increased reliance on Internet services supported by multi-tier applications where the typical three tiers are the web, middleware and database tier. In distributed systems, especially multi-tier applications, the fault in a component may manifest itself as an error, and then propagate to multiple services through the normal communication between the services. The error may remain undetected for an arbitrary length of time causing long error propagation chains. The error can propagate from one component to another and finally

manifest itself as a failure. The failure might be detected at a component distant from the originally faulty component. The availability of a system can be quantified as $MTTF/(MTTF+MTTR)$ (MTTF: Mean time to failure, MTTR: Mean time to recovery). There is enormous effort in the fault tolerance community to increase the reliability of components in a distributed system, thus increasing MTTF. There are also a growing number of efforts aimed at reducing MTTR [3]. An important requirement for this is to know which components to recover. This requires tracing back through the chain of errors to determine the component that originated the failure. This serves as the goal for our diagnosis protocol.

For the application and diagnosis model, consider that the application is comprised of multiple services communicating through standard protocols and externally observable messages. Example of such application services are web services and authentication services. The services themselves are comprised of multiple components, e.g. Enterprise Java Beans (EJBs) or servlets, and the interactions between these components are also externally visible. Separate from the application system, we have a monitoring system (the Monitor) that can observe the external interactions of the components but not their internal states. The Monitor initiates diagnosis when a failure is detected through an existing detection system. In this paper we use the existing detection system from our previous work [4].

In practical deployments, the Monitor may not observe the interaction between components of the application perfectly because of congestion or their relative network placement. This is particularly likely because the application as well as the Monitor is distributed with components spread out among possibly distant hosts. Next, any Monitor will have limited resources and may drop some message interactions from consideration due to exhaustion of its resources (e.g., buffers) during periods of peak load. Third, any diagnostic tests used by the Monitor might not be perfect. Finally, several parameters of the environment are not known deterministically and have

to be estimated at runtime. These include the ability of a component to stop the cascade of error propagation (error masking ability) and the unreliability of links between the application components and the Monitor. All these factors necessitate the design of a probabilistic diagnosis protocol, in which the root cause of the failure cannot be deterministically identified.

Our solution implemented in the Monitor rests on three basic techniques. First, the messages between the components are used to build a causal dependency structure between the components. Second, when a failure is detected, the causal structure is traversed (till a well-defined bound) and each component is tested using diagnostic tests. These diagnostic tests are not executed on the components directly but on the component state that had been deduced and stored at the Monitor. We decide against direct tests on the components because the state of the component may have changed since the time it propagated the error and the probing introduces additional stress on the component at a time when failures are already occurring in the system. Third, runtime observations are used to continually estimate the parameters that bear on the possibility of error propagation, such as unreliability of links and error masking capabilities. In our approach, the end goal of the probabilistic diagnosis process is to produce a vector of values called the *Path Probability of Error Propagation (PPEP)*. For the diagnosis executed due to a failure detected at component n , *PPEP* of a component i is the conditional probability that component i is the faulty component that originated the cascaded chain of errors culminating in n .

The basic structuring of an observer and an observed system is not new [4][5]. The problem of diagnosis of failures in networked environments comprised of black-box entities has also been studied by numerous researchers [6][7][8]. Some of these efforts however are aimed at easing the task of distributed debugging rather than accurate diagnosis of the faulty entity, some are offline approaches, some require accurate prior dependency information between the entities, and yet others need help from the application system through event generation. Our work aims to provide diagnosis of the faulty entities at runtime in a non-intrusive manner to the application.

We apply the diagnosis protocol to a three tier e-commerce system consisting of the Pet Store application deployed on the JBoss application server with the Tomcat web server as the front end and the MySQL database server at the backend. The application supports multiple kinds of browse-and-buy transactions that involve interactions between many components, where components are defined as servlets

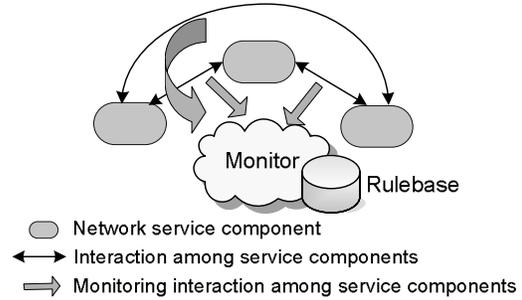


Figure 1. A monitoring system, (the Monitor) verifying the interactions between the service components

and EJBs. Through a modification to the JBoss containers, messages between the components are trapped and forwarded to the Monitor. We compare our approach to Pinpoint [7] in terms of accuracy and precision of diagnosis. Pinpoint uses statistical clustering of components with failed transactions to identify the faulty components. We inject errors in the application, where the errors may be due to a single component or interactions between multiple components. Our approach outperforms Pinpoint with the accuracy of the diagnosis improving from 20% to 100% over the Pinpoint algorithm for comparable precision values.

The rest of the paper is organized as follows. Section 2 presents the system model. Section 3 presents the probabilistic diagnosis approach. Section 4 presents the implementation and experimental test bed. Section 5 presents the experimental results. Section 6 reviews related work and section 7 concludes the paper. Table 1 provides a list of acronyms used in this paper.

Table 1. Table of acronyms

Acronym	Description
PPEP	Path Probability of Error Propagation
LC	Logical Clock
CG	Causal Graph
DT	Diagnostic Tree
STD	State Transition Diagram
EMC	Error Masking Capability
AG	Aggregate Graph

2. System Model and Background

There are two distinct systems—the Monitor and the application system. The Monitor obtains the protocol messages either through modification to the application’s middleware layer to forward the messages or by a passive snooping mechanism by the Monitor (see Figure 1). In either scenario the internal state of the components is not visible to the Monitor and they are treated as black-box for the diagnostic

process. The diagnostic process is triggered when a failure is detected.

2.1. Assumptions

We assume that components can fail arbitrarily, as long as the failures are observable in the message interaction between components. These failures could be caused by incorrect deployment, software bug, security vulnerability or performance problems to name a few. We follow the classical definition of faults being underlying defects that are triggered to become errors and some errors causing end-user visible failures. Errors can propagate from one component to another through the message exchanges between them.

The communication between the components can be asynchronous but the jitter on any given link between the components and the Monitor is bounded. This allows the Monitor to assign a temporal order in which messages occur in the application. For a non-zero jitter value there can be messages that the Monitor will determine to have occurred simultaneously. We assume that the Monitor maintains a logical clock for each observed component and it is incremented for each event – a send or receive message. The assumption required by the diagnosis protocol is that for an *S(ender)*-*R(eceiver)* communication, the variation in the latency on the *S-M(onitor)* channel as well as the variation in the sum of the latency in the *S-R* and *R-M* channels is less than a constant Δt , called the *phase*. If messages M_1 and M_2 , corresponding to two send events at S , are received at the Monitor at (logical) times t_1 and t_2 , it is guaranteed that the send event M_1 happened before M_2 if $t_2 \geq t_1 + \Delta t$. The communication channel is considered to be unreliable where message duplication, loss or conversion to another correct protocol message may happen.

2.2. Dependency Information

The Monitor performing diagnosis maintains a causal graph during the times that it is verifying the operation of the application protocol. Let us denote the *Causal Graph (CG)* at Monitor m by CG_m which is a graph (V, E) where (i) the set V contains all the components verified by m ; (ii) an edge or link e contained in E , between vertices $v1$ and $v2$ (which represent components) indicates interaction between $v1$ and $v2$ and contains state information about all observed message exchanges between $v1$ and $v2$ including the logical clock (*LC*) at each end. The state information includes a type of interaction and any arguments associated with that interaction. The links in the *CG* are also time-stamped with the local (physical) time at the Monitor where the link is created. An example of a *CG* created at the Monitor is given in

Figure 2 for the sequence of message exchange events shown with components A , B , C , and D . The number denotes the sequence of the messages. For example, for message ‘6’, the logical clock time at the sender is B.LC4. Since message ‘2’ is assigned a logical time value of B.LC2, it causally precedes message ‘6’. The *LC* time stamps helps obtain a partial order over the messages and hence causality. The order of the messages is the order seen by the Monitor which may be different from the order in the application because the communication links are asynchronous.

For a link to be completed in the *CG*, a matching is required between the sending and the receiving component’s messages. The link $A \rightarrow B$ will be matched once both the message sent by A and the corresponding message received by B is seen at the Monitor. The Monitor initially stores the messages in a *Temporary Links* table and moves the matched links to the *CG* when some trigger is met. As many links as can be matched are transferred to the *CG* while those that are not matched, but are within the phase from the latest message, are kept in the temporary links. Remaining links in the temporary links table are moved to the *CG* as unmatched links.

It is imperative to avoid the *CG* growing in an unbounded manner since this would lead to long delays in traversing the *CG* during diagnosis leading to high latency in diagnosis. However, complete purging of the information in the *CG* can cause inaccuracies during the diagnosis process. We aggregate the state information in the *CG* at specified time points and store it in an *Aggregate Graph (AG)*. The *AG* contains aggregate information about the protocol behavior averaged over the past. The *AG* is similar to *CG* in the structure i.e. a node represents a component and a link represents a communication channel. Unlike the *CG*, there is a single directed link between A and B for all the messages which are sent from A to B . The *AG* contains some node level information (such as, the node reliability) and some link level information (such as, the reliability of the link in the application system).

2.3. Diagnosis Tree

When a failure occurs, a *Diagnosis Tree (DT)* is constructed using the state information stored in *CG*. The *DT* formed for failure F at node D is denoted as DT_{FD} . The tree is rooted at node D and the nodes which have directly sent messages to node D are present at depth 1. Recursively, depth i consists of all the components which have sent messages to nodes at depth $(i-1)$. Since the *CG* is finite size, the tree is terminated when no causally preceding message is available in the *CG* after some depth k . The same component might appear several times in the tree at

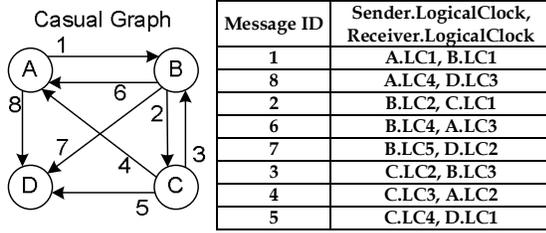


Figure 2. A sample causal graph. A, B, C and D exchange messages 1-8 among each other. The message ID indicates the causal order, i.e., message 1 precedes the rest of messages

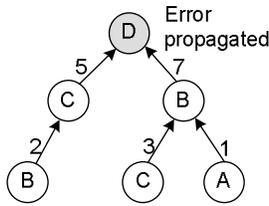


Figure 3. Sample DT for the CG in Fig. 2

various depths because it might have exchanged messages with various components at different points during the application run. Specifically, a component is represented as many number of times as the number of different states it has been in, while exchanging messages.

2.4. Diagnostic Tests

We assume the existence of *diagnostic tests* which operate on a component and are specific to that component and its state. We impose that the tests should only operate on the information already stored at the Monitor. These tests could be probabilistic in nature, implying that they may not be perfect. The specifics of these tests do not affect our probabilistic model. However, for our implementation, we employ a kind of tests called *causal tests*. A causal test has the format: $\langle Type \rangle \langle State1 \rangle \langle Event1 \rangle \langle Count1 \rangle \langle State2 \rangle \langle Event2 \rangle \langle Count2 \rangle$, where, *Type* could be one of {incoming, outgoing, hybrid} depending on the kind of messages being tested. The $(State1, Event1, Count1)$ forms the pre-condition to be matched, while $(State2, Event2, Count2)$ forms the post-condition that should be satisfied for the node to be deemed correct. The examination of *Event2* is done in an interval of time Δt (a phase) from *Event1*. The tuple (S, E, C) refers to the fact that the event *E* should have been detected in the state *S* at least counts *C* number of times.

The correctness rules can be created by examining the *state transition diagram (STD)* of the component and verifying the transitions or by observing some traces of the correct protocol operation. Additionally, rules corresponding to QoS requirements (such as, the number of accesses to the *SignOnEJB* in Pet Store must be restricted to 20 within a 1 sec time window) can be framed by the system administrators. Finally, rules for verifying security properties in the system (such as, the number of logins to Pet Store bounded by a threshold) can be set by security administrators. Rules therefore can be framed through a mix of automated and manual means. This is similar to the situation in all rule based systems, such as intrusion detection systems [9][10].

3. Probabilistic Diagnosis

The operation of the diagnosis protocol has two logical phases: (1) The diagnostic process that results in a set of nodes being diagnosed as the root cause of failure; (2) Information from the diagnostic process being used to update the parameters used later for diagnosis. The overall process is depicted in Figure 4. Let us first look at the diagnostic process. The goal of the diagnostic process is to calculate the probability of each node in the distributed system being faulty.

3.1. Path Probability of Error Propagation

The *DT* forms the basic structure on which the algorithm operates. The path from any node n_i to the root of the *DT* constitutes a possible path for error propagation, i.e. a fault present in n_i could have caused the root node to fail during operation. The probability of a path being the chain of error propagation is termed as the *Path Probability of Error Propagation (PPEP)*.

A sample *DT* created from the sample *CG* in Figure 2 is shown in Figure 3. Here the failure was manifested at node *D*. The numbers at the links correspond to the message IDs. The root of the tree is the failure node, i.e., *D*. Depth 1 consists of nodes *C* and *B* which have sent messages to *D* causally before the failure was detected. Here node *B* is repeated twice because the states of *B* in which $B \rightarrow C$ and $B \rightarrow D$ communication take place are different.

Definition: $PPEP(n_i, n_j)$, where $n_i \neq n_j$, is defined as the probability of node n_i being faulty and causing this error to propagate on the path from n_i to n_j , leading to a failure at n_j . The probability that the error is caused by the component where the failure is detected is denoted as $P(n_j) = PPEP(n_i, n_j)$, where $n_i = n_j$ and it is the root of *DT*. The relation between *PPEP* and $P(n_j)$ is

$$\sum_{n_i \in N} PPEP(n_i, n_j) < 1 - P(n_j)$$

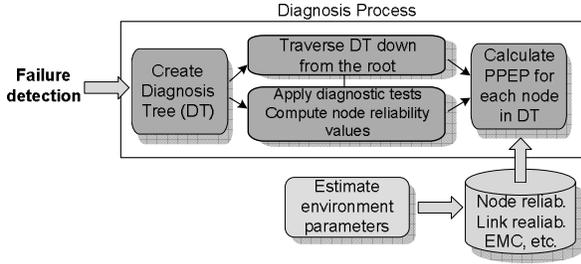


Figure 4. Schematic with the overall process flow of the diagnostic process

where n_j is the root of DT , and N is the set of all nodes in DT except the root node. This inequality is due to the fact that CG is truncated (as explained in section 2.2) and because DT is constructed from CG , it does not represent all the possible nodes and paths to the root node. $PPEP$ depends on the following parameters:

(1) **Node reliability** – The node reliability (n_r) is a quantitative measure of the component corresponding to the node being correct. The $PPEP$ for a given node is proportional to $(1 - n_r)$. This node reliability is obtained by running the diagnostic tests on the state of the entity. A set of predetermined tests are performed, each of which yields a ‘0’ (test flags an error) or a ‘1’ (success). If the entire set of tests is denoted by R and a subset of tests which yield ‘1’ be denoted by R' , we define coverage $c(n) = |R'|/|R|$, assuming all tests have equal weights. For the first time the diagnosis is triggered, the node reliability is equal to $c(n)$. During the entire execution of the application, multiple failures cause multiple diagnosis procedures to execute. Each time the diagnosis is performed, node reliabilities (in AG) corresponding to all of the nodes in the DT are updated. In [11] it is explained how node and link reliabilities are updated.

(2) **Link Reliability** – In simple terms the link reliability between two nodes n_i and n_j denoted as $l_{r(i,j)}$ measures the number of received packets by the receiver over the number of packets actually sent. $PPEP$ can be considered proportional or inversely proportional to the link reliability between nodes, depending on the nature of the application being observed. On the one hand, we can assume (as we have done in our current design) that a reliable link increases the probability of the path being used for propagating errors. However, we can also think an unreliable link as the source of unexpected inputs to subsequent components leading to a higher value of $PPEP$. In our framework, $PPEP$ is proportional to link reliability because we consider that, in the majority of applications, errors are more likely to be propagated in the presence of reliable links. The Monitor does not diagnose a link as the root cause of a failure, rather it

diagnoses application components. Thus, if in reality, an unreliable link initiates the cascading chain of errors, then the first component that is affected by the link error will be diagnosed. Since the Monitor does not detect link failures, we do not inject failures in links for our experiments. The link reliability is maintained for each edge in the AG . Note that since the Monitor is only observing the system, the errors within the Monitor in observing the messages also may erroneously affect the link reliability. These errors cannot be distinguished in our framework. The Monitor deduces the link reliability through observing the fraction of successful message transmissions over a particular link.

(3) **Error masking capability (EMC)** – The error masking capability (e_m) quantifies the ability of a node to mask an error and not propagate it to the subsequent link on the path in the DT toward the root. The EMC of an entity depends on the type of error, e.g., syntactical or semantic errors. Additionally, a node may have different error masking capabilities depending on the type of message being processed and forwarded, e.g., if there is an *off-by-one* bug in the counter check on the number of simultaneous JDBC connections, it will mask the errors when the number of JDBC connections is one more than the maximum threshold. The EMC of node C in Figure 3 is denoted by $e_m(C)$ and is a function of *message type* and *error type*. The $PPEP$ for a given node is inversely proportional to the EMC values of nodes in the path since the intermediate nodes are less likely to have propagated the error to the root node.

With the DT in Figure 3, $PPEP(C, D) = (1 - n_r(C)) \cdot l_{r(C,D)}$, $PPEP(B, D) = (1 - n_r(B)) \cdot l_{r(B,C)} \cdot (1 - e_m(C)) \cdot l_{r(C,D)}$. For a general path P consisting of nodes $n_1, n_2 \dots n_k$ with link $l_{r(i,j)}$ between nodes n_i and n_j , the $PPEP(n_1, n_k)$ for a failure detected at node n_k (root node in the corresponding DT) is given by

$$PPEP(n_1, n_k) = (1 - n_r(n_1)) \cdot l_{r(1,2)} \cdot (1 - e_m(n_2)) \cdot l_{r(2,3)} \dots \cdot l_{r(i,i+1)} \cdot (1 - e_m(n_{i+1})) \cdot l_{r(i+1,i+2)} \dots \cdot (1 - e_m(n_{k-1})) \cdot l_{r(k-1,k)}$$

We consider that all rules cannot be matched at runtime because that would impose unnecessary overhead and would not be useful in most executions (when there is no failure). The first component that failed a test does not necessarily implicate the component that sent it, other factors are to be considered, such as, how reliable were the links between that component and the component at which the failure was ultimately detected. This is because the distance from the root is not a matter simply of the number of links on the DT . Also, the tests are not perfect and cannot therefore be trusted to indict a component by themselves.

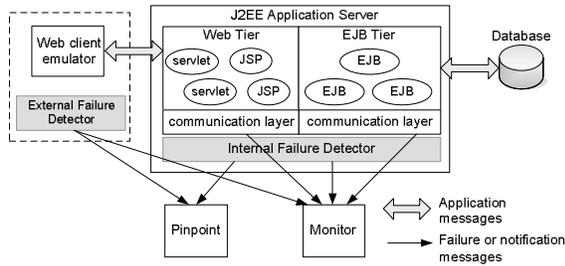


Figure 5. Logical Topology of the Client and Server for the Experiments

4. Experimental Testbed

4.1. Application

We use for our evaluation Pet Store (version 1.4), a sample J2EE application developed under the Java BluePrints program at Sun Microsystems [12]. It runs on top of the JBoss application server [13] with MySQL database [14] as the back-end providing an example of a 3-tier environment. Figure 5 depicts the application topology for the experiments. The Pet Store application is driven by a web client emulator which generates client transactions based on sample traces. The web client emulator is written in Perl using lynx as the web browser. For the mix of client transactions, we mimic the TPC-WIPSo [15] distribution with equal percentage of browse and buy interactions. The servlets and the EJBs are considered as components in our experiments and these serve as the granularity level at which diagnosis is done. This design choice is based partly on the fact that in JBoss a faulty servlet or an EJB can be switched out at runtime for a correct one. We identify a total of 56 components in the application.

We consider a *web interaction* to be a complete cycle of communication between the client emulator and the application, as it is defined by the TPC Benchmark WIPSo specification [15]. Examples of web interactions could be entering the Welcome page or executing a Search. A *transaction* is a sequence of web interactions. An example of a transaction by a user who is searching and viewing information about a particular product is: *Welcome page* → *Search* → *View Item details*. For our experiments we created a total of 55 different transactions. A *round* is a permutation of these 55 transactions modeling different user activities that occur on the web store. Within a round, transactions are executed one at a time. Two transactions are considered to be non-unique if they use exactly the same components, neglecting the order in which the components are used. Thus, a transaction that comprises: *Welcome, Search, Search* is not unique

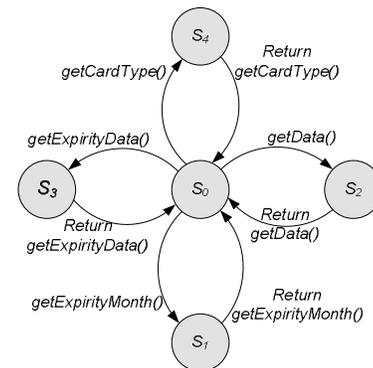
with respect to another that comprises: *Welcome, Search*. There are 41 unique transactions in the set of 55 transactions that we use. Although 55 is not an exhaustive set of possible transactions in the application, the chosen set exercised a wide variety of web-interactions and between them, touched all the components of Pet Store. We note that the results presented here depend on the exact set of transactions used to exercise the system.

4.2. Monitor configuration

The Monitor is provided an input of state transition diagrams for the verified components and causal tests used during calculation of *PPEP* values. The size of the causal graph is bounded at 100 links. Figure 6 shows an example *STD* for *CreditCardEJB* used by the Monitor in our experiments. A start state S_0 signifies a no request state. If a request for processing is received from another component, the state of the EJB moves from S_0 accordingly. With the *STD*, we have some simple causal tests which can be derived from the *STD* itself. As explained in section 2.4, causal tests are dependent on the state and event of the component. The exhaustive list of *STDs* and rules used for the experiments here is provided in [16].

4.3. Pinpoint Implementation

Pinpoint serves as a valid point of comparison with the Monitor since both systems have the same focused goal (diagnosis, as opposed to say performance debugging as in [6] with diagnosis being a side issue) and have the same target application model (black-box or gray-box application and passive observation of the application for diagnosis). Importantly, Pinpoint represents a recent state-of-the-art development ([7][17]) and has been well explained and demonstrated on an open source application (compare



```
S0 getData 1 S2 return getData1 1
S0 getExpiryMonth 1 S1 return getExpiryMonth 1
S0 getExpiryData 1 S3 return getExpiryData 1 1
S0 cardType 1 S4 return cardType1 1
```

Figure 6. An example STD for CreditCardEJB and some Causal Tests

to Magpie [18] where the application is not available to us), and its algorithms are not dependent on a large set of parameters (compare to the machine learning approach in [19][20] where several statistical distributions would have to be assumed).

We implement the Pinpoint algorithm (as explained in [7]) for comparison with the Monitor’s diagnosis approach. Pinpoint diagnosis algorithm requires as input a dependency table—a mapping of which components each transaction depends on. This is in contrast to the Monitor approach, where such dependency information does not have to be determined *a priori* and fed into the system before execution. Instead the Monitor deduces the dependencies through runtime observations as described in section 2.2. For Pinpoint, when transactions are executed, their failure status is determined by the failure detectors. A table (called the input matrix) is then created with the rows being the transactions, the first column being the failure status, and the other columns being the different components. If a cell $T(i, l)$ is 1, it indicates transaction i has failed. If a cell $T(i, j)$ ($j \neq l$) is 1, it indicates transaction i uses the component j . Pinpoint correlates the failures of transactions to the components that are most likely to be the cause of the failure.

A crucial point for the accurate operation of Pinpoint is that the transactions should be diverse enough, i.e., use distinct non-overlapping components. Two transactions T_1 and T_2 are called distinct with respect to a set of components $\{C_1, C_2, \dots, C_k\}$ if there is no overlap between these columns for T_1 and T_2 , i.e., when T_1 ’s row has a 1 in any of these columns, T_2 ’s row has a zero, and vice-versa. Pinpoint as described by the authors in [7] is an offline approach. For comparison with the Monitor, we convert it into an online protocol. We incrementally feed the transactions and their corresponding failure status as they occur in the application, rather than waiting for all the transactions in a round to be completed before executing Pinpoint. To provide a comparable platform between the Monitor and Pinpoint, we keep the testbed identical to that in [7]—same client, web server, application server (with identical components), and database server. Pinpoint is sensitive to the transactions used; however, [7] is silent on the list of used transactions and we were unable to obtain them.

We created an internal and an external failure detector as in [7] to provide failure status of transactions to Pinpoint and the Monitor.

4.4. Fault Injection

We perform fault injection into the components of the Pet Store application (i.e., Servlets and EJBs). We

choose a set of 9 components called *target components* (see Figure 12) consisting of six EJBs and three servlets for fault injection. We use four different kinds of fault injection as in [7], i.e., declared exception, undeclared exception, endless call and null call. The injected faults affect node reliability and *EMC* for the *PPEP* calculations.

The internal detector is more likely to detect the declared and the undeclared exceptions, and the null calls while the external detector is more likely to detect the endless call. For a given round only one target component is injected. We use 1-component, 2-component and 3-component triggers. In a 1-component trigger, every time the target component is touched by a transaction, the fault is injected in that component. In a 2-component trigger, a sequence of 2-components is determined and whenever the sequence is touched during a transaction, the last component in the transaction is injected. This mimics an interaction fault between two components, and, in the correct operation of a diagnosis protocol, both components should be flagged as faulty. The 3-component fault is defined similarly.

5. Results

5.1. Performance Metrics

We use *precision* and *accuracy* as output metrics as in the Pinpoint work to enable a comparison. A result is accurate when all components causing a fault are correctly identified. For example, if two components, A and B , are interacting to cause a failure, identifying both would be accurate. Identifying only one or neither would not be accurate. However, if the predicted fault set (by the diagnosis algorithm) is $\{A, B, C, D, E\}$ and the fault was in components $\{A, B\}$, then the accuracy is still 100%. Precision captures the non-idealness in this case. Precision is the ratio of the number of faulty components to the total number of entities in the predicted fault set. In the above example, the precision is 40%. Components $\{C, D, E\}$ are false positives. Lower precision implies high false positives. There is a tension between accuracy and precision in most diagnosis algorithms. When the algorithm is sensitive, it generates highly accurate results, but also causes a large number of false alerts reducing precision. Pinpoint uses the UPGMA clustering algorithm and varying the size of the faulty cluster varies the precision and accuracy. In the Monitor, after the diagnosis algorithm terminates, an ordered list of components is produced in decreasing order of *PPEP*. We define the predicted fault set as the top k components in the ordered output list. We vary k to obtain different accuracy and precision values.

5.2. Single Component Faults

In single component faults, the fault injection trigger consists of a single component. If a transaction touches the target component then one of the four kinds of faults (chosen randomly) is injected and the injection remains permanent for the remainder of the round. First, let us consider the effect of varying cluster size on the performance of Pinpoint. The total number of injections for these results is 36—9 target components for injection and all 4 types of injection done on each component. The averaged results for accuracy and precision are plotted in Figure 7 (the bars show 90% confidence interval). As the size of the cluster increases, we see an increase in the accuracy which is intuitive because at some point the failure cluster includes all the components that are actually faulty. Beyond that, increase in cluster size does not impact the accuracy. As the cluster size increases, the precision increases to a maximum value and then decreases thereafter. The increase occurs till all the faulty components are included in the failure cluster. Thereafter, increasing the cluster size includes other non-faulty components and thus brings down the precision. The maximum value of precision occurs when all the faulty components are included in the failure cluster. However the precision is still poor (less than 10%). This is explained by the observation that for the transactions in the application, there is tight coupling between multiple components. Whenever the entire set of tightly coupled components does not appear together as a fault trigger, which is the

overwhelming majority of the injections, the precision suffers. The amount of tight coupling between the components is showed in Figure 12. We emphasize that if we were to hand pick transactions such that they are distinguishable with respect to the target components, then the performance of Pinpoint would improve. Two transactions T_i and T_j are indistinguishable with respect to a set of components $\{C_1, C_2, \dots, C_k\}$ if the columns of T_i in the input matrix corresponding to these components are identical to that of T_j . Figure 8 shows the variation of Accuracy with False Positives for Pinpoint and the Monitor. This is averaged across the 36 injections for the presented results. For 1-component faults, Pinpoint has high false positives rates but the accuracy eventually reaches 1. In contrast the Monitor has a much higher accuracy keeping a low false positive rate. The Monitor's accuracy also reaches 1 but at a much lower value of false positives (0.6) as compared to Pinpoint (> 0.9). The latency of detection in our system is very low. Thus, the faulty component is often at the root of the *DT* in the Monitor. Since error propagation is minimized, the *PPEP* value for the faulty entity is high causing it to be diagnosed by the Monitor. This explains the high accuracy for the Monitor. However, Pinpoint's algorithm does not take advantage of the temporal information—the temporal proximity between the component where detection occurs and the component that is faulty. As a consequence its accuracy suffers relative to that of the Monitor. Note that we do not have a figure corresponding to Figure 7 for the Monitor since its performance does not depend

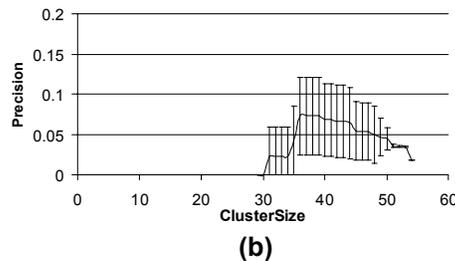
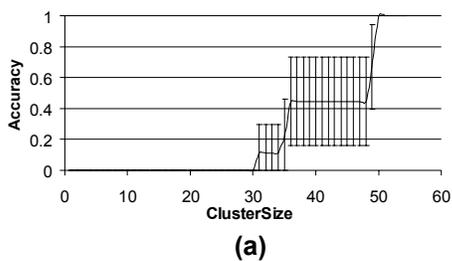


Figure 7. Single component fault injection: Variation of (a) accuracy and (b) precision with cluster size in Pinpoint

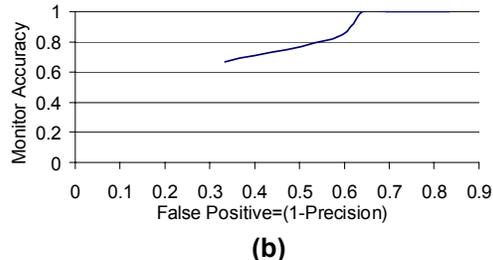
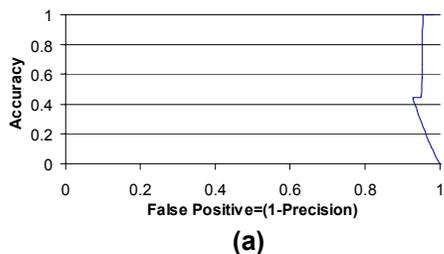
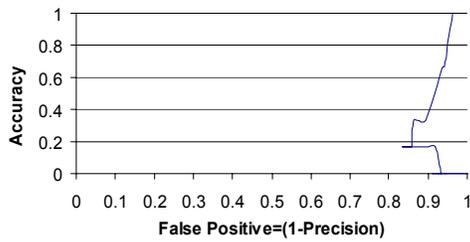
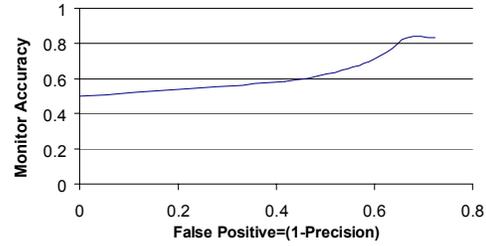


Figure 8. Single component fault injection: Performance of (a) Pinpoint and (b) Monitor. Both can achieve high accuracy but Pinpoint suffers from high false positive rates

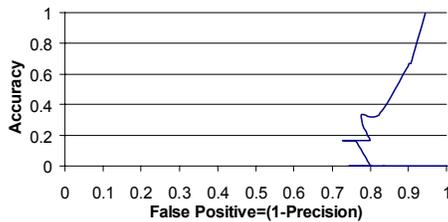


(a)

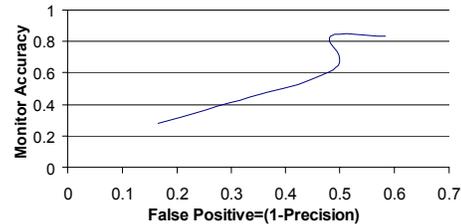


(b)

Figure 9. 2-component fault injection: Performance of (a) Pinpoint and (b) Monitor. Performance of the Monitor declines and Pinpoint improves from the single component fault, but the Monitor still outperforms Pinpoint



(a)



(b)

Figure 10. 3-component fault injection: Performance of (a) Pinpoint and (b) Monitor. Performance of the Monitor declines and Pinpoint improves from the single and two component faults, but the Monitor still outperforms Pinpoint

on cluster size.

Notice that in Pinpoint, for a given value of false positives, two different accuracy values are achieved since a given precision value is achieved for two different cluster sizes (Figure 7(b)). Since accuracy is a monotonically increasing plot with cluster size (Figure 7(a)), the different cluster sizes give two different accuracy values. For a given data point, the accuracy is either 100% (when the single injected component is included by Pinpoint in the diagnosed set) or 0%, which is then averaged across the total number of experiments. These discrete values explain the large confidence intervals.

5.3. Two Component Faults

The 2-component fault injection results are shown in Figure 9. Pinpoint results improve in terms of the false positives implying higher precision. This is attributed to the fact that Pinpoint's clustering method works better if the failing transactions are better distinguishable from the successful transactions. Recall distinguishable is discussed in the context of components. A 2-component fault includes two components as the trigger and going from one component to two components increases the distinguishability of transactions. Consider transaction T_1 and T_2 both of which use component C_1 (i.e., the trigger in a single component fault injection). However, for a two component fault injection with trigger as $\{C_1,$

$C_2\}$, the transactions T_1 and T_2 will be distinguishable as long as both T_1 and T_2 do not use C_2 . Thus, say T_1 uses $\{C_1, C_2\}$ and T_2 does not use C_2 . Then only T_1 will fail and T_2 will not, leading to the diagnosis (considering simplistically that these are the only transactions and components) of C_1-C_2 as the faulty entities. In contrast, the Monitor results although, still significantly better than Pinpoint, suffer in the 2-component fault injection. One can see that accuracy reaches a maximum of only 0.83 compared to 1.00 in 1-component injection. The number of times in a round the trigger for the 2-component fault is hit is lower than for the single component fault. Each detection causes an execution of the diagnosis process and each execution of the diagnosis process updates the parameters of the causal graph away from an arbitrary initial setting toward an accurate set of values. Thus, for the 2-component faults, the Monitor gets less opportunity for refining the parameter values and consequently the *PPEP* calculation is not as accurate as for the single component faults. This explains the decline in performance of the Monitor for the 2-component faults.

5.4. Three Component Faults

The 3-component fault injections show even better results for Pinpoint with the maximum average precision value touching 27%. This is again attributed to the fact that more number of components causes

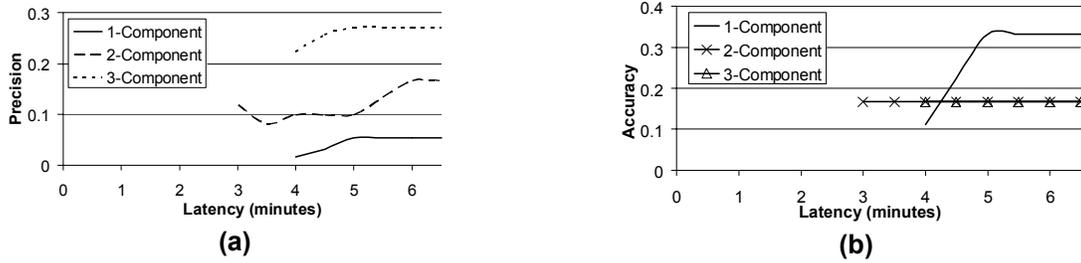


Figure 11. Single component fault injection: Variation of (a) precision and (b) accuracy and precision with latency for Pinpoint in single component fault injection. Higher latency means higher number of transaction data points and Pinpoint’s performance improves monotonically

selected transactions to fail leading to a better performance by the clustering algorithm. The Monitor again outperforms Pinpoint by achieving higher accuracy at much lower false positives (see Figure 10). The Monitor’s performance again declines compared to the 2-component faults due to the same reason pointed in the previous section (the number of diagnoses for the 3-component trigger is less than that for the 2-component trigger).

5.5. Latency

In its online incarnation, Pinpoint takes as input the transactions and corresponding failure status every 30 seconds during a round. It runs the diagnosis for each of these snapshots taken at 30 second intervals, terminating when the round is complete and Pinpoint executes on the entire input matrix corresponding to all the 55 transactions. The latency plots (see Figure 11) show that after 3.5 minutes the accuracy and precision of Pinpoint increase with latency. Pinpoint’s performance is only defined for the time points after the first failure has been injected and detected. For our experiments, this happens at and beyond 3 minutes. To the left of this point, both accuracy and precision are undefined since Pinpoint does not predict any

component to be faulty before a failure is detected.

We define the latency of diagnosis for the Monitor as the time delay from the receipt of the detector alert which marks the beginning of the diagnosis till the *PPEP* ordered list is generated. The Monitor has an average latency of 58.32 ms with a variance of 14.35 ms, aggregated across all three fault injection campaigns.

5.6. Behavior of Components

The Pet Store application has some components which are tightly coupled (see Figure 12), i.e., they tend to be invoked together for the different transactions supported by the application. We have noted earlier that tight coupling negatively impacts Pinpoint’s clustering algorithm. For our experiments, we inject failures in 9 components and here we consider how tightly coupled these components are with the other components in Pet Store. *AddressEJB* is tightly coupled with 4 components implying that it always occurs with these 4 components in all the 55 transactions in our experimental setup. Pinpoint cannot distinguish between sets of components that are tightly coupled and thus reports all the tightly coupled components as faulty even though in reality only a subset of these may be faulty. This is the fundamental reason why its precision is found to be low in all our experiments. To counter this problem, one can synthetically create transactions that independently use different components (as noted by the authors themselves in [7]). However, for an application like Pet Store, components are naturally tightly coupled and thus generating such synthetic transactions is a difficult task. Also even if we could devise such “unnatural” transactions that would make components distinguishable, it cannot be assumed that such transactions will be created by users in the system.

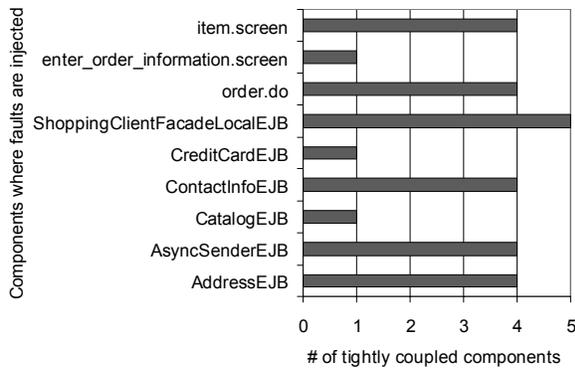


Figure 12. Number of tightly coupled components for Pet Store

6. Related Work

White box systems: The problem of diagnosis in distributed systems can be classified according to the

nature of the application system being monitored – white box where the system is observable and, optionally, controllable; and black box where the system is neither. The Monitor system falls in the latter class. White box diagnostic systems often use event correlation where every managed device is instrumented to emit an alarm when its state changes [21][22]. By correlating the received alarms, a centralized manager is able to diagnose the problem. Obviously, this depends on access to the internals of the application components. Also it raises the concern whether a failing component’s embedded detector can generate the alert. This model does not fit our problem description since the target system for the Monitor comprises of COTS components, which have to be treated as black-box. White box diagnosis systems that correlate alarms have been proposed also in the intrusion detection area [23].

Debugging in distributed applications: There has been a spurt of work in providing tools for debugging problems in distributed applications – performance problems [6][7][18], misconfigurations [24], unexpected behavior [24], etc. The general flavor of the approaches in this domain is that the tool collects trace information at different levels of granularity and the collected traces are automatically analyzed, often offline, to determine the possible root causes of the problem. For example, in [6], the debugging system performs analysis of message traces to determine the causes of long latencies. The goal of these efforts is to deduce dependencies in distributed applications and flag possible root causes to aid the programmer in the manual debug process, and not to produce automated diagnosis.

More recent work has produced powerful tools for debugging of distributed applications. In [25], the 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, i.e., no message is received before it has been sent. This work stops short of automated diagnosis. Some other mechanism, not described in the paper, is responsible for taking the replayed events and determining the root cause. There are several other offline tools that aid diagnosis, such as tools for data slicing [26], and backtracking, but they all require manual effort in diagnosing the faulty components.

Network diagnosis: Diagnosis in IP networks is addressed in Shrink [28]. This tool used for root cause analysis of network faults models the diagnosis problem as a Bayesian network. It specifically diagnoses inaccurate mappings between IP and optical layers. The work in [29] studies the effectiveness and practicality of Tree-Augmented Naive Bayesian

Networks (or TANs) as a basis for performing offline diagnosis and forecasting from system-level instrumentation in a three-tier network service. The TAN models are studied to select combinations of metrics and thresholds values that correlate with performance states of the systems (compliance with Service Level Objectives). This approach differs from the Monitor approach in the sense that it relies on monitoring performance metrics rather than diagnosing the origin of the problem over a set of possible components.

Automated diagnosis in COTS systems: Automated diagnosis for black-box distributed COTS components is addressed in [30]. The system model has replicated COTS application components, whose outputs are voted on and the replicas which differ from the majority are considered suspect. This work takes the restricted view that all application components are replicated and failures manifest as divergences from the majority. In [31], the authors present a combined model for automated detection, diagnosis, and recovery with the goal of automating the recovery process. However, the failures are all fail-silent and no error propagation happens in the system, the results of any test can be instantaneously observed, and the monitor accuracy is predictable.

In none of the existing work that we are aware of is there a rigorous treatment of the impact of the Monitor’s constraints and limited observability on the accuracy of the diagnosis process. There are sometimes statements made on this without supporting reasoning – for example, in [6], it is mentioned that drop rates up to 5% do not affect accuracy of the diagnosis.

7. Conclusion

In this paper we presented an online diagnosis system called the Monitor for arbitrary failures in distributed applications. The Monitor passively observes the message exchanges between the components of the application and at runtime, performs a probabilistic diagnosis of the component that was the root cause of a detected failure. The Monitor is compared to the state-of-the-art diagnosis framework called Pinpoint. We tested the two systems on a 3-tier Java-based e-commerce system called Pet Store. Extensive fault injection experiments were performed to evaluate the accuracy and precision of the two schemes. The Monitor outperformed Pinpoint particularly in precision, though its advantage narrowed for interaction faults. As part of future work we are looking at diagnosis in high throughput network streams. In these streams, the Monitor may have to decide to drop some parts of a stream. We are looking into intelligent decision making to maintain a high

accuracy. We are also investigating machine learning based diagnosis in the presence of uncertain information.

8. References

- [1] META Group, Inc., "Quantifying Performance Loss: IT Performance Engineering and Measurement Strategies", November 22, 2000. Available at: <http://www.metagroup.com/cgi-bin/inetcgi/jsp/displayArticle.do?oid=18750>.
- [2] FIND/SVP, 1993, "Costs of Computer Downtime to American Businesses," At: www.findsvp.com.
- [3] A. Brown and D. A. Patterson, "Embracing Failure: A Case for Recovery-Oriented Computing (ROC)," 2001 High Performance Transaction Processing Symp., Asilomar, CA, October 2001.
- [4] G. Khanna, P. Varadharajan, and S. Bagchi, "Self Checking Network Protocols: A Monitor Based Approach," In Proc. of the 23rd IEEE Symp. on Reliable Distributed Systems (SRDS '04), pp. 18-30, October 2004.
- [5] M. Zulkernine and R. E. Seviora, "A Compositional Approach to Monitoring Distributed Systems," IEEE International Conference on Dependable Systems and Networks (DSN'02), pp. 763-772, Jun 2002.
- [6] M. K. Aguilera, J. C. Mogul, J. L. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance debugging for distributed systems of black boxes," Proc. of the 19th ACM Symp. on Operating Systems Principles (SOSP), pp. 74-89, 2003.
- [7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: problem determination in large, dynamic Internet services," Intl. Conf. on Dependable Systems and Networks (DSN), pp. 595-604, 2002.
- [8] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox, "Autonomous Recovery in Componentized Internet Applications," Cluster Computing Journal, Vol. 9, Number 1 (February 2006).
- [9] K. Bhargavan, S. Chandra, P. J. McCann, and C. A. Gunter, "What Packets May Come: Automata for Network Monitoring," In ACM SIGPLAN Notices, vol. 36, no. 3, pp. 206-219, 2001.
- [10] "Snort Flexible Response Add-On," Available at: http://cerberus.sourceforge.com/~jeff/archives/snort/sp_response2/
- [11] G. Khanna, I. Laguna, F. Arshad and S. Bagchi, Technical Report, School of Electrical and Computer Engineering, Purdue University, May 2007, "<http://docs.lib.purdue.edu/ecetr/354/>".
- [12] Pet Store J2EE Application: <http://java.sun.com/blueprints/code/index.html>.
- [13] JBoss Application Server: <http://labs.jboss.com>.
- [14] MySQL: Open Source Database: www.mysql.com.
- [15] TPC- Benchmark: <http://www.tpc.org/tpcw>.
- [16] Addendum to SRDS '07 submission-data files and communication: www.ece.purdue.edu/~sbgchi/Papers/srds07_add.pdf
- [17] G. Candea, E. Kiciman, S. Kawamoto, and A. Fox, "Autonomous Recovery in Componentized Internet Applications," Cluster Computing Journal, Vol. 9, Number 1 (February 2006).
- [18] P. Barham, R. Isaacs, R. Mortier, and D. Narayanan, "Magpie: online modeling and performance-aware systems " at the 9th Workshop on Hot Topics in Operating Systems (HotOS IX), pp. 85-90, 2003.
- [19] I. Rish, M. Brodie, and S. Ma, "Intelligent probing: A cost-efficient approach to fault diagnosis in computer networks," IBM Systems Journal, vol. 41, no. 3, pp. 372-385, 2002.
- [20] I. Rish, M. Brodie, M. Sheng, N. Odintsova, A. Beygelzimer, G. Grabarnik, and K. Hernandez, "Adaptive diagnosis in distributed systems," IEEE Transactions on Neural Networks, vol. 16, no. 5, pp. 1088-1109, 2005.
- [21] B. Gruschke, "Integrated Event Management: Event Correlation Using Dependency Graphs," at the 10th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM), pp. 130-141, 1998.
- [22] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo, "A coding approach to event correlation," Intelligent Network Management, no., pp. 266-277, 1997.
- [23] F. Cuppens and A. Mieke, "Alert correlation in a cooperative intrusion detection framework," Proceedings of the 2002 IEEE Symp. on Security and Privacy, May 12-15, 2002.
- [24] H. J. Wang, J. Platt, Y. Chen, R. Zhang, and Y.-M. Wang, "PeerPressure for automatic troubleshooting," at the Proc. of the joint international conference on Measurement and modeling of computer systems, New York, NY, USA, pp. 398-399, 2004.
- [25] D. Geels, G. Altekar, S. Shenker, and I. Stoica, "Replay Debugging for Distributed Applications," USENIX Annual Technical Conference, pp. 289-300, 2006.
- [26] X. Zhang, R. Gupta, and Y. Zhang, "Precise dynamic slicing algorithms," ICSE, pp. 319-329, 2003.
- [27] S. T. King, G. W. Dunlap, and P. M. Chen, "Debugging operating systems with time-traveling virtual machines," USENIX Annual Technical Conference, pp. 1-15, 2005.
- [28] S. Kandula, D. Katabi, J. Vasseur, "Shrink: A Tool for Failure Diagnosis in IP Networks," ACM SIGCOMM Workshop on mining network data (MineNet-05), Philadelphia, PA, August 2005.
- [29] I. Cohen, M. Goldszmidt, T. Kelly, S. Julie, J. Chase, "Correlating instrumentation data to system states: A building block for automated diagnosis and control", Operating System Design and Implementation (OSDI), Dec 2004.
- [30] A. Bondavalli, S. Chiaradonna, D. Cotroneo, and L. Romano, "Effective fault treatment for improving the dependability of COTS and legacy-based applications," Dependable and Secure Computing, IEEE Transactions on, vol. 1, no. 4, pp. 223-237, 2004.
- [31] K. R. Joshi, W. H. Sanders, M. A. Hiltunen, R. D. Schlichting, "Automatic Model-Driven Recovery in Distributed Systems," At the 24th IEEE Symp. on Reliable Distributed Systems (SRDS'05), pp. 25-38, 2005.