Problem Determination using Dependency Graphs and Run-time Behavior Models

Manoj K. Agarwal[1], Karen Appleby[2], Manish Gupta[1], Gautam Kar[2], Anindya Neogi[1], and Anca Sailer[2]

[1] IBM India Research Laboratory,
New Delhi, India
{manojkag, gmanish, anindya_neogi}@in.ibm.com
[2] IBM T.J. Watson Research Center, Hawthorne, NY, USA
{gkar, applebyk, ancas}@us.ibm.com

Abstract. Key challenges in managing an I/T environment for e-business lie in the area of root cause analysis, proactive problem prediction, and automated problem remediation. Our approach as reported in this paper, utilizes two important concepts: dependency graphs and dynamic runtime performance characteristics of resources that comprise an I/T environment to design algorithms for rapid root cause identification in case of problems. In the event of a reported problem, our approach uses the dependency information and the behavior models to narrow down the root cause to a small set of resources that can be individually tested, thus facilitating quick remediation and thus leading to reduced administrative costs.

1 Introduction

A recent survey on Total Cost of Operation (TCO) for cluster-based services [1] suggests that a third to half of TCO, which in turn is 5-10 times the purchase price of the system hardware and software, is spent in fixing problems or preparing for impending problems in the system. Hence, the cost of problem determination and remediation forms a substantial part of operational costs. Being able to perform timely and efficient problem determination (PD) can contribute to a substantial reduction in system administration costs. The primary theme of this paper is to show how automatic PD can be performed using system dependency graphs and run-time performance models.

The scope of our approach is limited to typical e-business systems involving HTTP servers, application servers, messaging servers, and databases. We have experimented with benchmark storefront applications, such as TPC-W bookstore [2]. The range of problems in distributed applications is very large, from sub-optimal use of resources, violation of agreed levels of service (soft-faults), to hard failures, such as disk crash. In a traditional management system, problem determination is related to the state of components at system level (e.g., CPU, memory). Thus, a monitored system component that fails, notifies the management service, which manually or

automatically detects and fixes the problem. However, when a transaction type "search for an item in an electronic store" shows a slowdown and violates user SLA (in this paper, we do not distinguish between SLA and SLO (Service Level Objective)), it is often an overwhelming and expensive task to figure out which of the many thousands of components, supporting such a transaction, should be looked at for a possible root cause. In this paper, we focus on the soft-faults within the e-business service provider domain. An approach towards tackling this complex area is combining the run-time *performance modeling* of system components with the study of their *dependencies*. The next section will provide a short summary of the notion of dependencies and dependency graphs, reported in detail in previous publications [4].

The main thesis of this paper is that PD applications can use the knowledge provided by dependency graphs and resource performance models to quickly pinpoint the root cause of SLA or performance problems that typically manifest themselves at the user transaction level. The monitoring data, say response time, is collected from individual components and compared against thresholds preset by a system administrator. Each time the monitored metric exceeds the threshold, an alert event is sent to a central problem determination engine, which correlates multiple such events to compute the likely root cause. Thresholds are hard to preset, especially in the event of sudden workload changes, and their use often results in spurious events. The primary contribution of this paper is that, we dynamically construct response time baselines for each component by observing its behavior. When an SLA monitor observes an end-to-end transaction response time violation due to some degradation inside the system, the individual components are automatically ranked in the order of the violation degree of their current response time level with their constructed good behavior baseline and of their dependency information. A system administrator can then scan the limited set of ranked components and quickly determine the actual root cause through more detailed examination of the individual components. The net gain here is that the administrator would need to examine far fewer components for the actual root cause, than conventional management approaches.

In this paper, we describe the creation of simple performance models using response time measurement data from components, end-to-end SLA limits, and component dependency graphs. We show how, given end-to-end SLA violations, these dynamic models, in combination with dependency graphs, can be used to rank the likely root cause components.

The management system has an architecture designed in three tiers, as shown in Fig. 1. The first tier consists of monitoring agents specific to server platforms. These agents can interface with the monitoring APIs of the server platforms, extract component-wise monitoring data and send them to the second tier. In the second tier, the online mining engine (OME) performs dependency extraction (if required), weighs the extracted dependencies, and stores them in a repository. The accurate dependency data may be provided through transaction correlation instrumentation, such as ARM [3]. Otherwise it is extracted by mining techniques in OME [4][5], from aggregate monitoring data. A standardized object-oriented management data modeling technology called Common Information Modeling (CIM) [6] is used when storing the dependency information in a database. The third tier comprises management

applications, for example the PD application to be described in this paper, which uses the CIM dependency database.

The rest of the paper is structured as follows: Section 2 provides a short background on dependency analysis and dependency graphs. Section 3 outlines some of the more popular tools and approaches for performing PD to establish the relevance of our work to this area. In Section 4 we describe our algorithms for resource behavior modeling and we show how they can be used for PD in Section 5. Section 6 presents the prototype environment on which our PD technique is being applied and tested. We conclude the paper in Section 7 with a summary and a discussion of our on-going and future work in this domain.

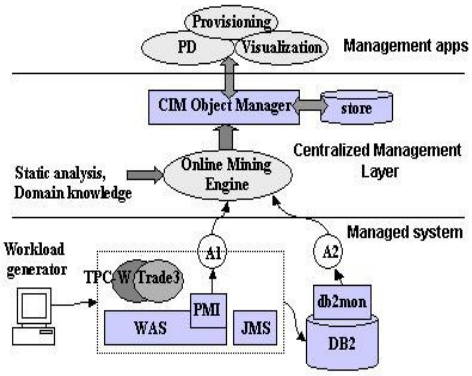


Fig. 1. Management system architecture

2 Background

This section presents an overview of the general concept of dependencies as applied to modeling relationships in a distributed environment.

Consider any two components in a distributed system, say *A* and *B*, where *A*, for example, may be an application server, and *B* a database. In the general case, *A* is said to be dependent on *B*, if *B*'s services are required for *A* to complete its own service. One of the ways to record this information is by means of a directed graph, where A and B are represented as nodes and the dependency information is represented by a directed arc, from A to B, signifying that A is dependent on B. A is referred to as the *dependent* and B as the *antecedent*. A weight may also be attached to the directed edge from *A* to *B*, which may be interpreted in various ways, such as a quantitative measure of the extent to which *A* depends on *B* or how much *A* may be affected by the non-availability or poor performance of *B*, etc. Any dependency between *A* and *B* that arises from an invocation of *B* from *A* may be synchronous or asynchronous.

There are different ways in which dependency information can be computed. Many of these techniques require invasive instrumentation, such as the use of ARM [3]. The algorithms that we have designed and implemented [5] do not require such invasive changes. Instead they infer dependency information by performing statistical correlation on run time monitored data that is typically available in a system. Of course, this approach is not as accurate as invasive techniques, but our experiments show that the level of accuracy achieved is high enough for most management applications, such as problem determination/root cause analysis, that can benefit by using the dependency data. Here, accuracy is a measure of how well an algorithm does in extracting all existing dependencies in a system. An additional consequence of a probabilistic algorithm, such as ours, is that false dependencies may be recorded which could mislead a PD application into identifying an erroneous root cause. We have devised a way of minimizing such adverse effects by ensuring that our probabilistic algorithms attach low weights to false dependencies. Thus, if all the antecedents of a dependent component were ranked in order of descending weights in the dependency graph, a PD application, while traversing this graph would be able to identify the root cause before encountering a false dependency with low weight. A measure of how disruptive false dependencies are in a weighted dependency graph is *precision* [4]. Simply stated, a dependency graph with high precision is one where the false dependencies have been assigned very low weights. In the next section we highlight some of the PD systems that are available today and point out their relevance to our work.

3 Related Work

Problem Determination (PD) is the process of detecting misbehavior in a monitored system and locating the problems responsible for the misbehavior. In the past, PD techniques have mainly concentrated on network [7], and system [9] level fault management. With the emerging Internet based service frameworks such as e-commerce sites, the PD challenge is how to pinpoint application performance root causes in large dynamic distributed systems and distinguish between faults and their consequences.

In a traditional management system, PD is related to the state of components at system level (e.g., CPU, memory, queue length) [9]. In application performance analysis, the starting point for choosing the metrics for detecting performance problems is the SLA. In our scenario, we consider a response time based SLA and characterize the system components in terms of their response time to requests triggered by user transactions. Our solution addresses the case of ARM enabled systems as well as legacy systems, and relies on agents (both, ARM agents and native agents) to collect monitoring data.

The classical approach to constructing models of the monitored components is one that requires detailed knowledge of the system [11][12]. As such models are difficult to build and validate. Most approaches use historical measurements and least-squares regression to estimate the parameters of the system components [13]. Diao *et al.* use

the parameters in a linear model and the model generation is only conducted once for a representative workload, experimentally showing that there is no need to rebuild the model once the workload changes [14]. We generate the behavior characteristics of the monitored components based on historical measurements and statistical techniques, distinguishing between the good behavior model and the bad behavior model. Furthermore, while many efforts in the literature address behavior modeling of individual components, e.g., Web Server [14], DB2 [15], we characterize the resources' behavior keeping in mind the end-to-end PD of the application environment as a whole.

Most PD techniques rely on alarms emitted by failed components to infer that a problem occurred in the system [19]. Brodie *et al.* discuss an alternate technique using synthetic transactions to probe the system for possible problems [16]. Steinder *et al.* review the existing approaches to fault localization and also presents the challenges of managing modern e-business environments [8]. The most common approaches to fault localization are AI techniques (e.g., rule-based, model-based, neural networks, decision trees), model traversing techniques (e.g., dependency-based), and fault propagation techniques (e.g., codebook-based, Bayesian networks, causality graphs). Our solution falls in the category of model traversing techniques. Bagchi *et al.* implement a PD technique based on fault injection, which may not be acceptable in most e-business environments [17]. Chen *et al.* instrument the system to trace request flows and perform data clustering to determine the root cause set [21]. Our technique uses dynamic dependencies inferred from monitored data without any extra instrumentation or fault injection.

4 Behavior Modeling Using Dependency Graphs

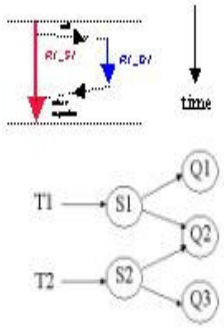
We assume an end-user SLA with an end-to-end response time threshold specified for each transaction type. An SLA monitor typically measures the end-to-end response time of a transaction, but it has no understanding of how the transaction is executed by the distributed application on the e-business system. Hence, when an SLA limit for a transaction type is exceeded, the monitor has no idea about the location of the bottleneck within the system. In this section, we describe how one can construct dynamic thresholds for the internal components by observing their response time behavior.

4.1 Monitoring

A threshold is an indicator of how well a resource is performing. In most management systems today, thresholds are fixed, *e.g.*, an administrator may set a threshold of x seconds for the response time of a database service, meaning that if the response is over x, it is assumed that the database has a problem and an alert should be issued. We introduce the concept of *dynamic thresholds*, which can be changed and adjusted on a regular basis through our behavior modeling, thus accommodating changes in operating conditions, such as application load. The good behavior model or dynamic

threshold of a component is constructed based on two inputs: response time samples obtained through the monitoring infrastructure and a resource dependency graph. A typical real-life monitoring infrastructure provides only aggregate information, such as average response time and access counts of components etc. In our earlier work [4][5] we have shown how such aggregate monitoring information can be used to construct aggregate dependency graphs. As shown in Fig. 2, an aggregate graph captures the dependency of a transaction type on resources aggregated over multiple transaction instances.

Our technique of dynamic threshold computation uses an aggregate monitoring infrastructure and aggregate dependency graphs. Such graphs may even have imperfections, such as false and/or missing dependencies. In an extended research report, we show how our PD algorithm deals with such shortcomings [20]. Our dynamic threshold computation technique currently uses data from HTTP Server logs, WAS Performance Monitoring Infrastructure (PMI), and DB2 Snapshot API. We assume that the same aggregate monitoring APIs have also been used for dependency graph construction.




```
1. read response time RT for component C
2. lookup graph to create T,
        set of transactions that depend on C
3. if any element of T is in "bad" state
   1.    bad_avg(C) = αbad_avg(C) + (1-α)RT
         bad_N(C) = bad_N(C) + 1
   else
         good_avg(C) = αgood_avg(C) + (1-α)RT
4. if(bad_N(C) > β)
         compute severity(C) = bad_avg(C) / good_avg(C)
5. when problem is diagnosed reset
                  bad_N(C) = bad_avg(C) = 0
```

Fig. 2. Aggregate graph and model-builder logic

4.2 Behavior Modeling

The goal of behavior modeling is to construct a dynamic threshold of a component, such that when an end-to-end problem is detected, the current response time samples from the component may be compared with its dynamic threshold.

A transaction type can have two states, henceforth called "good" or "bad", corresponding to when they are below or above their SLA limits, respectively. Similarly, each system component should also have a good state or a bad state depending on whether they are the cause of a problem or are affected by a problem elsewhere. In a traditional management system, a hard-coded threshold is configured on each individual component. A component is in bad state if its response time is beyond the threshold else it is in good state. Each component in bad state sends an event to a central event correlation engine, which determines the likely root cause

based on some human generated script or expert rule base. This approach results in a large number of events from various components. Besides, it is very difficult and error prone for the system administrator to configure a threshold for a component without extensive benchmarking experience.

Our management system uses average response time samples from the components to build their bad or good state performance models. A key feature of our system is that it uses the dependency graph to classify response time samples from a component into bad and good state, instead of hard-coded thresholds on individual components. The classification rule states that if *any parent transaction* of a component is in bad state when the response time sample is obtained, then the sample is classified as "bad" and added to the bad behavior model of the component, otherwise it is added to its good behavior model. The good behavior model is an average of the good response time values and also serves as the dynamic threshold.

Fig. 2 shows the dependency graph of transactions T1 and T2. S1 sometimes accesses Q1 and sometimes Q2. When a response time sample from query Q2 is obtained, the model-builder logic checks the current state of T1 as well as T2. Only the SLA monitor can modify the state of T1 and T2. If T1 *and* T2 are in good state, the sample is added to the good model of Q2. If either of them is bad because the fault lies in any of the component in the sub-tree of the bad transaction, the sample is added to the bad model of Q2. The problem determination logic is invoked after a few samples of the bad model are obtained. Thereafter, the bad and good models of each component are compared and the components are ranked as described in Section 5. In our current implementation, a good or bad model is simply the average of the distribution of good or bad values, respectively. Fig. 2 shows the pseudo-code for the model-builder logic.

The good model of a component is persistent across problem phases, i.e., it is never forgotten and more samples make it more dependable for comparison against a bad model. The bad model samples are typically unique to the particular type and instance of the problem. Hence the bad models are forgotten after each problem is resolved. We assume that problems are not overlapping, i.e., there is only one problem source and independent problem phases do not overlap.

In our current implementation, the cumulative response time of a component obtained from the monitoring infrastructure is used as the model variable. This response time includes the response time of the child components. For example, the average response time of S1 includes average response times of Q1 and Q2, as illustrated in Fig. 2. Thus, if a bottleneck is created at Q1, Q1 as well as S1's response time behavior models are affected.

The cumulative time is effective in identifying a faulty path in the dependency tree, but, in many cases, is not adequate in pinpointing the root-cause resource. We are working on an enhanced approach, where the model variable can be changed to capture the local time spent at a component, excluding the response time of the children. This approach will be reported in a later paper.

5 Problem Determination

In this section we discuss how components may be ranked, so that a system administrator may investigate them in sequence to determine the actual bottleneck. In normal mode of operation each component computes a dynamic threshold or a good behavior model. When a problem occurs at a component, the dependent transactions are affected and all components that are in the transaction's sub-tree start computing a bad behavior model. The components that do not build a bad behavior model in this phase, i.e., those that do not belong to a sub-tree of any affected transaction type, are immediately filtered out. The next step is to rank all the components in the sub-tree of an affected transaction.

Each component is assigned a severity value, which captures the factor by which the bad model differs from the good behavior model or dynamic threshold of the component. Since a model in the current implementation is a simple average of the distribution of samples, a simple ratio of the bad model average to the dynamic threshold represents the severity value. Fig. 5 shows a graph with severity values computed per node when the problem is at Q2. For example, for component Q2, the bad model is 105.2 times the dynamic threshold. The un-shaded nodes are not considered because they do not have a bad model and are assigned a default severity of 0.

The shaded components are sorted based on their severity value as shown in the first ranking. Besides the root cause component, say Q2, the components that are on the path from transaction T1 to Q2, such as S1 and S2, have high severity values because we use the cumulative response time as the model variable and not the local time spent at a component. Bad models are computed for other nodes in the subtree, such as Q1 and Q3, but their bad model is very close to their good model because they do not lie on the "bottleneck path". Thus, in the first ranking we prune and order the components in the subtree so that only nodes, which are on the "bottleneck path" are clustered on top. However, this is not enough to assign the highest rank to the root cause node. There is no guarantee that a parent of a root cause node, such as S1, is not going to have higher severity value. For example, in the first ranking in Fig. 5, Q2 appears after S1.

It is possible to reorder the components further based on dependency relationship and overcome the drawback of using the cumulative response time for modeling. Given the severity values of the shaded nodes, we apply a standard 2-means clustering algorithm [18] to divide the set into "high severity set" and "low severity set". In our experience, the severity values of the affected and root cause components are much higher than the unaffected components. For example, the components in Fig. 5 are divided into high severity set: {S1, S2, Q2} and low severity set: {Q1, Q3}. In the second ranking, if a parent and child are both in the high severity set and the parent is ranked higher than the child in the first ranking, then their rankings are swapped. The assumption here is that the high severity of the parent has resulted from the high severity of the child. The assumption holds if there is a single fault in the system and the transactions are synchronous. Since S1 and Q2 are in the same set, they are reordered and Q2 is picked as the highest rank. Thus a system administrator

investigating the components will first look at Q2 before any other component. The efficiency of our technique is defined by the rank assigned to the root cause node.

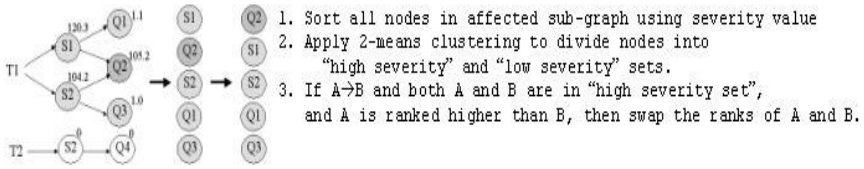


Fig. 3. Ranking logic

Building performance models and subsequent PD is unaffected by the presence of the false dependencies or by the aggregate representation of dependency graphs. In the interest of space, a complete proof is presented in an extended research report [20].

6 Experimental Evaluation

In this section we present the experimental results to demonstrate the efficiency of our PD technique using behavior models and dependency graphs.

The experimental setup is shown in Fig. 1. The OME is used to extract dependencies between servlets and SQLs in the TPC-W application installed on WAS and DB2. The TPC-W bookstore application is a typical electronic storefront application [2] consisting of 14 servlets, 46 SQLs and a database of 10,000 books. The extracted dependency graph is stored in the CIM database and used by the PD application. The monitoring data is gathered through agents and used by the PD application to build performance models. An SLA monitor (not shown in the figure) intercepts all HTTP requests and responses at the HTTP server. These responses are then classified as 'good' or 'bad' based on the SLA definition. We set individual SLA thresholds for all 14 transaction types in the TPC-W application as our user level SLA definitions. Problems are injected into the system through a problem injector program (not shown in the figure), that periodically locks randomly chosen servlets on WAS or database tables on DB2 with an on-off duty cycle for the injection period, to simulate higher response times for the targeted servlets or tables. The TPC-W code is instrumented to implement the servlet level problem injection. Once we lock the table or servlet, all transactions based on that particular table or servlet slow down and we see an escalation in the response times of the corresponding transactions at the user level and thus violation of the SLAs. We have 10 tables in the DB2 holding data for the TPC-W application and 14 servlets. Thus we can inject problems at 24 different locations in the system. We log these injected problems in a separate log file as the *ground truth*. We then use this ground truth information to compute the efficiency of our PD technique. The efficiency is measured in terms of average accuracy and average rank of the root cause in the ordered list of probable components, where the averaging is performed over multiple problem injections. Accuracy is the measure of

finding an injected problem in the list of probable root causes discovered by our PD algorithm. The rank measure of the root cause is the position the root cause component occupies in the ordered list of probable root causes. If the injected problem lies in the n^{th} position from the top of listed root causes, it is assigned rank n. The success of our PD technique is determined by how close the average accuracy and the average rank of the root cause are to 100% and rank 1, respectively.

Table 1. Effect of dependency information with servlet and table level problems

Load	Graph Type	Avg. Acc (%)	Avg Rank1	List Size
40	ARM	100	1.3	2.1
	Mining	100	1.1	2.7
	Instant	100	1.3	5.5
80	ARM	100	1.5	3.2
	Mining	100	1.4	4.0
	Instant	100	1.5	7.1
120	ARM	100	1.8	3.7
	Mining	100	1.6	5.0
	Instant	100	1.3	10.5

Table 2. Effect of dependency information on table level problems

Load	Graph Type	Avg. Rank1	Avg Rank2	%age chnge
40	ARM	2.4	1.7	20
	Mining	2.1	1.6	24
	Instant	1.2	1.2	0
80	ARM	1.4	1.0	28
	Mining	1.7	1.5	15
	Instant	2.0	1.1	45
120	ARM	1.8	1.5	16
	Mining	1.2	1.2	0
	Instant	1.3	1.3	2

Dependency information used by the PD technique can be obtained by three means. An accurate and precise graph may be obtained through ARM instrumentation. A graph with some false dependencies may be obtained through the online mining techniques presented in [4][5]. We take a TPC-W bookstore graph with 100% accuracy and 82% precision extracted at a load of 100 simultaneous customers. This graph, labeled "mining" in Tables 1 and 2, is used as a more imprecise graph. Finally, we also consider a bottom-line case in which historical dependency knowledge is not used but classification is done based on instantaneous information. For example, in the TPC-W application, transactions are synchronous. Thus if a component B occurs when transaction A is active, we consider that as a dependency. This case, termed "instant" in Table 1 and Table 2, contains all the possible dependencies including much more spurious ones compared to "mined" graphs. We investigate the effect of the quality of the dependency information on the efficiency of our PD technique. We inject a set of problems sequentially over time with sufficient gaps between the problems so that the system recovers from one problem before experiencing another. We also vary the system load to observe its effect on behavior modeling and PD. Load is the number of simultaneous customers active in the system sending URL requests to the TPC-W application. At the load of 120 customers, the load generator sends around 300 URL requests/minute. We run these experiments over the duration of 2 hours each during which we inject randomly chosen 12 different problems out of set of 24 problems. Each problem is injected 5 to 10 times and the average accuracy and average rank are computed over all injected problems. Table 1 summarizes the results of our experiments.

We see that accuracy of our PD algorithm is always 100%. It means that we can always find the injected problem in our list of suspected root causes. There are total 60 different components (14 servlets and 46 SQLs). The last column "list size" shows the average number of components selected for ranking, which decreases as the quality of the dependency information increases. Thus the quality of the dependency information definitely helps in reducing the set of components that are considered (the shaded nodes in Fig. 3). However, it does not impact the ranking to a significant extent (see proof in [21]). The rank of the root cause, using this technique, in which all the components are sorted based on severity, lies between 1 and 2. This means that the root cause is almost always the first or the second component in the ordered list. Besides, the behavior modeling and PD based on the dynamic thresholds, is also not heavily impacted by load. The average rank of the root cause in this approach increases only marginally, as load increases.

We also investigate the application of dependency graph to improve the first ranking. Here we inject problems only at table level so that we can observe the effect of swapping ranks between servlets and antecedent SQLs. In Table 2, "Avg Rank2" is the average rank of the root cause after applying the dependency information on the first ranking. In most cases, the average rank of the root cause is improved in the second ranking. In the cases where the percentage improvement is not significant enough, their "Avg Rank1" is already close to the minimum possible rank. More experimentation is needed to find out the effect of load and graph type on the percentage improvement.

7 Conclusion

In this paper, we have presented our research in the area of Problem Determination for large, distributed, multi-tier, transaction based e-business systems. The novelty of our approach, as compared to others reported in the literature, is that we use a combination of resource dependency information and resource behavior models to facilitate the rapid isolation of causes when user transactions manifest unacceptably slow response time.

One of the drawbacks of our current approach is that, in some cases, when a user transaction misbehaves, we are able to narrow down the root cause to a set of resources that support the transaction, but may not be able to identify the offending resource. This is because resource behavior models are inclusive, i.e., a dependent resource's model includes the effects of its antecedents. As ongoing work we are looking at enhancing our approach to constructing models that better reflect the performance of individual resources, thus providing a better framework for root-cause analysis. One approach is to compute a resource's good behavior by capturing its individual contribution to a transaction's end-to-end response time. In addition, we are investigating how our technique can provide a reliable basis for problem prediction, through the observation of trends in the variation of resource behavior. We are extending our approach for proactive problem prediction, before the problem manifests as a user level SLA violation.

References

1. Gillen A., Kusnetzky, McLaron S., The role of linux in reducing cost of enterprise computing, IDC white paper, January 2002.
2. TPCW: Wisconsin University, http://www.ece.wisc.edu/~pharm/tpcw.shtml.
3. ARM: Application Response Measurement, www.opengroup.org/zsmanagement/arm.htm
4. M. Gupta, A. Neogi, M. Agarwal, G. Kar, Discovering dynamic dependencies in enterprise environments for problem determination, Proceedings of 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management, October 2003.
5. M. K. Agarwal, M. Gupta, G. Kar, A. Neogi, A. Sailer, Mining activity data for dynamic dependency discovery in e-business systems, under review for eTransactions on Network and Service Management (eTNSM) Journal, Fall 2004.
6. CIM: Common Information Model, http://www.dmtf.org/standards/standard_cim.php.
7. R. Boutaba, J. Xiao, network management: state of the art, IFIP World Computer Cogress2002,http://www.ifip.tugraz.ac.at/TC6/events/WCC/WCC2002/papers/Boutaba.pdf
8. M. Steinder, A.S. Sethi, The present and future of event correlation: A need for end-to-end service fault localization, Proc. SCI-2001, 5th World Multiconference on Systemics, Cybernetics, and Informatics, Orlando, FL (July 2001), pp. 124-129.
9. Y. Ding, C. Thornley, K. Newman, On correlating performance metrics, CMG 2001.
10. A. J. Thadhani, Interactive User Productivity, IBM System Journal, 20, p 407-423, 1981.
11. K. Ogata, Modern control engineering, Prentice Hall, 3rd Edition, 1997.
12. D. A. Menascé, D. Barbara, R. Dodge, Preserving QoS of e-commerce sites through self-tuning: a performance model approach, Proceedings of the 3rd ACM conference on Electronic Commerce, 2001.
13. S. Parekh, N. Gandhi, J. L. Hellerstein, D. Tilbury, T. S. Jayram, J. Bigus, Using control theory to achieve service level objectives in performance management, 2003.
14. Y. Diao, J. L. Hellerstein, S. Parekh, J. P. Bigus, Managing web server performance with autoTune agents, IBM Systems Journal 2003.
15. Y. Diao, F. Eskesen, S. Froehlich, J. L. Hellerstein, L. F. Spainhower, M. Surendra, Generic online optimization of multiple configuration parameters with application to a database server, DSOM 2003.
16. M. Brodie, I. Rish, S. Ma, N. Odintsova, Active probing strategies for problem diagnosis in distributed systems, in Proceedings of IJCAI 2003.
17. S. Bagchi, G. Kar, J. L. Hellerstein, Dependency analysis in distributed systems using fault injection: application to problem determination in an e-commerce environment, DSOM 2001.
18. C.M. Bishop, Neural networks for pattern recognition, Oxford, England: Oxford University Press, 1995.
19. K. Appleby, G. Goldszmidt, M. Steinder, Yemanja, A layered fault localization system for multi-domain computing utilities, in IM 2001.
20. M. Agarwal, K. Appleby, M. Gupta, G. Kar, A. Neogi, A. Sailer, Problem determination and prediction using dependency graphs and run-time behavior models, IBM Research Report, RI04004.
21. M.Y. Chen, E. Kıcıman, E. Fratkin, A. Fox, E. Brewer, Pinpoint: PD in large, dynamic internet services, International Conference on Dependable Systems and Networks (DSN'02), 2002.