RC24286 (W0706-065) June 15, 2007
Computer Science

IBM Research Report

Performance Problem Prediction in Transaction Based e-Business Systems

Manoj K Agarwal, Gautam Kar, Ruchi Mahindru, Anindya Neogi, Anca Sailer
IBM Research Division
Thomas J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598

Performance Problem Prediction in Transaction Based e-Business Systems

Manoj K. Agarwal, Gautam Kar, Ruchi Mahindru, Anindya Neogi, and Anca Sailer

***Abstract*— Key areas in managing environments that implement e-commerce systems are problem prediction, root cause analysis, and automated problem remediation. Anticipating SLO violations by proactive (rather than reactive) problem determination is particularly important since it can significantly lower the business impact of application performance problems. The approach investigated here is based on two important concepts: dependency graphs and dynamic runtime performance characteristics of resources that comprise an I/T environment. The authors show how one can calculate and use the extent to which supporting resources for a transaction contribute to the end-to-end SLOs for that transaction. An important aspect of this process is the classification of user transactions based on the profile of their resource usage, enabling one to set appropriate thresholds for different classes. Combined with the complete or semi-complete dependency information, our approch confines the scope of potential root causes to a small set of components, thus enabling efficient performance problem anticipation and quick remediation.**

***Index Terms*— SLA management, automatic threshold management, proactive problem determination, system dependencies.**

I. Introduction

APPLICATION Service Level Objective (SLO) management in an I/T infrastructure for on-line e-commerce is a challenging task. At the user level, it involves guaranteeing end-to-end response time and throughput for the different types of transactions that comprise a users interaction with the e-commerce system. In order to ensure such guarantees, the management system needs to monitor the flow and response time of transactions, identify the areas of bottleneck that might limit throughput or introduce delays, and execute remedial procedures that correct such problems.

Anticipating SLO violations by proactive (rather than only reactive) problem determination could significantly lower the impact of application performance problems. A 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 the operational costs. Being able to perform timely and proactive problem determination can contribute to a substantial reduction in system administration costs.

Manoj K. Agarwal and Anindya Neogi are with IBM India Research Lab, New Delhi. Gautam Kar, Ruchi Mahindru, and Anca Sailer are with IBM T.J. Watson Research Center, New York. The corresponding author is Anca Sailer, ancas@us.ibm.com.

This paper presents an architecture that enables proactive application problem determination based on the general concept of resource dependency relationships [2], [3]. The user transactions are supported by these resources. Hereafter, a transaction SLO is specified at the application level and is called Transaction-SLO or T_SLO. In a reactive approach, the antecedent resources are examined to pinpoint the root cause when T_SLO violations occur. In our proactive approach, the potential root causes of imminent problems are detected when T_SLO notifications of an imminent violation are received, before the actual T_SLO violation occurs. The main theme of this paper deals with the design and implementation of a management system for proactively determining performance problems in an e-commerce service provider domain.

Fig. 1 depicts an e-commerce management system designed on three tiers. The first tier consists of monitoring agents (M) specific to server platforms such as HTTP servers, application servers, and databases. In the second tier, the Management Service performs dependency extraction, assigns weights to the extracted dependencies, and stores them in a repository. A standardized object-oriented management data repository technology called Common Information Modeling (CIM) [4] is used to store the dependency information. When a problem such as transaction slowdown (depicted by step a in Fig. 1) manifests itself at the user level, the problem may lie anywhere in the system: the HTTP server, the network, the web application server, the backend database etc. In a traditional management system, problem determination is related to the state of components at the system level (e.g., CPU, memory) and application related problems that are often difficult to pinp

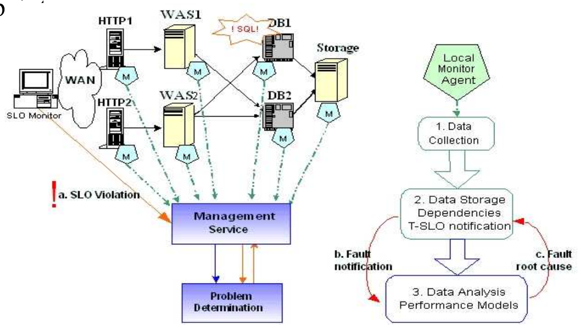


Fig. 1. Monitoring Architecture for a Standard e-commerce system

Our research addresses this issue by combining application dependency information with individual resource behavior models, in the third tier of Fig. 1. The work reported here complements our research reported in [5]. It is a major extension of the way we use the resource operation models and compute thresholds to perform proactive root cause analysis, as detailed in Section 4. Additionally, it is described in Section 5 of the analysis of the dependency graph accuracy impact on the quality of the problem determination.

A resource dependency graph is a valuable tool to isolate root causes of detected problems, or analyze the impact of issues on other resources. A pictorial representation of the notion of transaction dependency is shown in Fig. 2. The resources that comprise the system are grouped into various service layers, e.g., access, application, database, etc. Dependency relationships may exist between components located in different systems, such as between a Web Application Server (WAS) instance in one server and a database instance in another server, as well as between components that belong to the same logical system boundary, such as servlets and EJBs within an individual WAS instance.

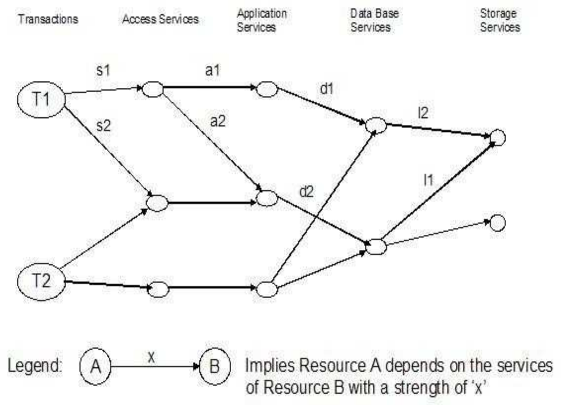


Fig. 2. Dependency Relationships in an e-commerce Application

In this paper, we assume that the dependency graph has already been constructed and is available [2], [3]. The research results reported here concentrate on the third tier of Fig. 1. The first contribution of this paper is an algorithm that uses monitored information to experimentally build performance models. These models are used to categorize transactions into classes, and then derive resource operational thresholds based on the SLOs of the classes. The second contribution is an algorithm that uses the dynamically derived operational thresholds and the dependency information to anticipate performance problems and perform proactive root cause analysis based on the resources' performance.

The rest of the paper is structured as follows: Section 2 provides a short survey of related work. In Section 3, we present the details of performance behavior modeling for resources and show how user transactions can be classified based on the behavior of the resources that they depend on. Section 4 describes our problem determination algorithms, while the results of the experimental validation are in Section 5. We conclude the paper in Section 6 with current ideas for future research work.

II. Related Work

In a traditional management system, problem determination (PD) is related to the state of resources at the system level (e.g., CPU, memory, queue length). In application performance analysis, the usual starting point for detecting performance problems is the SLO. Response time has been considered since the late seventies as an essential metric for quantifying system productivity, and tools have been constantly developed to accurately detect delays in various components of typical distributed systems. In our scenario, we consider a response time based SLO, while we characterize the system components in terms of the time locally spent at each of them by requests triggered by user transactions.

The solutions proposed in the literature with respect to predictions at application level, mainly focus on performance predictions from historical data [6], [7] or solving queuing network models [8]–[10]. The goal of these approaches is either to provision the system appropriately [6], or to limit the traffic access [8] in order to satisfy the SLO for the incoming traffic. These solutions do not consider unexpected performance values that may occur due to failures in the system that can jeopardize the capacity planning estimations or the limits imposed on the arriving traffic.

Intensive work has been done in the network research community to predict network failures. The proposed solutions are based on TCP related data [11], [12] or MIB variables [13], [14] which present highly non-stationary characteristics making it difficult to accurately model the dynamics of the system without large amounts of input information and processing requirements [15]. In [13], the authors use the GLR algorithm which is known to be computationally complex [16], while [14] uses Bayesian networks which fails to capture propagation of abnormal behavior through dependent network entities. The proactive network failure determination solutions represent a rich collection of predictive techniques tailored for the network traffic conditions and network problems. Proactive failure determination at application level needs to evaluate the application performance and its related middleware specific traffic conditions and failures, while using the network experience to avoid techniques that proved complex or inefficient.

Steinke [17] presents a survey of existing products and monitoring techniques used in application performance management and PD with the ultimate goal of getting a grip on performance at the business process level. The author remarks that the standardized management software of the Distributed Management Task Force [18] and the Open Group's Application Response Measurement working group [19] does not apply to all applications that need to be monitored. Therefore, legacy applications still require a different technology to be monitored and managed. The most widely adopted solutions that provide precise and reliable monitoring measurement and do not require application modification are the agent-based solutions. Our solution addresses the case of ARM enabled systems as well as legacy systems, and relies on agents (ARM or native agents) to collect monitoring data.

Steinder [20] reviews the existing approaches to fault localization and also presents the challenges of managing modern e-business environments. 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. Many of these techniques are event-based. Alarms relying on static dependencies between system components [21], [22] or on symptom-fault maps [23], [24] may be analyzed to perform PD. Unfortunately, since systems implementing Internet-based application and services change constantly, it is very difficult to accurately maintain such system knowledge and consequently use these PD techniques. In [22], the authors achieve PD based on fault injection and dynamic dependency information generated by Active Dependency Discovery (ADD) [25]. ADD builds the system dependency graph by individually perturbing the system resources during a test phase, while PD uses fault injection at run-time, which may not be acceptable in most e-business environments. Kiciman [26] has proposed a solution to overcome the limitation of static dependencies by tracing the request as they travel through the system. The authors perform data clustering analysis over a large number of requests to determine the expected combinations of components and most likely root causes. When a trace does not fit the expected pattern, the misbehaving resource is considered the root cause. Our technique uses dynamic dependencies inferred from monitored data, without requiring invasive changes to provide trace information. We use such dependency information and resource behavior models to troubleshoot SLO related problems.

III. Resource Operation Modeling based on Dependency Graphs

In most management systems today, the SLO monitor, which observes the transaction's end-to-end response time and notifies the management service when the specified thresholds are violated, cannot identify resource bottlenecks. In this section, we explain the key features of our management system which models resources' behavior, such that when an end-to-end T_SLO notification occurs, the current state of related components may be compared with their dynamic threshold, leading to a method for systematically identifying the root cause of potential T_SLO violations. We assume hereafter two types of SLO monitor notifications:

1) T_SLO violation notification used when the SLO monitor observes a transaction end-to-end response time exceeding the SLO-threshold, and
2) T_SLO imminent violation notification used when the SLO monitor observes a transaction end-to-end response time approaching the SLO-threshold by less than a percentage δ_{SLO}.

A. Input data

Our component performance indicator is a threshold, called component_SLO (c_SLO) for each transaction type. c_SLO reflects the time spent by a transaction at a specific resource. It is constructed based on three inputs: (i) resource response time samples, (ii) components dependency graph, and (iii) T_SLO monitor notifications. (i) The response time samples are obtained through the available monitoring infrastructure. (ii) Our behavior modeling technique applies to ARM [19] enabled systems where the dependency information is explicitly available, as well as to legacy systems, where the dependency graph generation is done as we showed in Section 1, using aggregate monitoring data. Fig. 3 illustrates such a dependency graph generated over multiple transaction instances. (iii) We use the T_SLO monitor notifications to distinguish the monitoring data generated under normal operating conditions from the monitoring data altered by failure. Thus, the c_SLOs are computed in such a way so as to only reflect the normal behavior of resources.

The monitors used for gathering response time samples for operational behavior model construction and specific data for dependency extraction are Web Server API, WAS Performance Monitoring Infrastructure (PMI), and DB2 Snapshot API.

B. Modeling Metric

In a traditional management system, the thresholds configured for each component are typically hard-coded limits for the components expected response time. If the response time is beyond the limit, the component sends an event to a management server which tries to correlate this event with others received around the same time, guided by expert rules, in order to identify the root cause. This approach often requires examining a large number of events from various components. A drawback of this approach is that the response time of components is a cumulative metric. Thus, components without any problem may send threshold violation notifications as they include in their response time, the response time of defective child components. As an example, Fig. 3 shows the dependency graph of two transactions T1 and T2. The response time of resource S1 that serves transaction T1 includes the response time of S1's children Q1 and Q2, as illustrated in Fig. 4. Hence, if Q1 experiences a service slow down, both Q1 and S1's response time values are affected and reported, although only Q1 is failing.

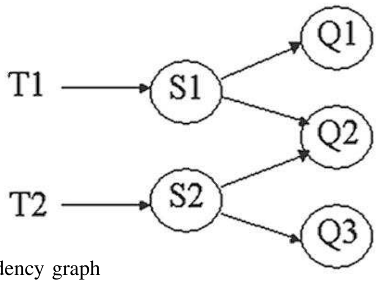


Fig. 3. Dependency graph

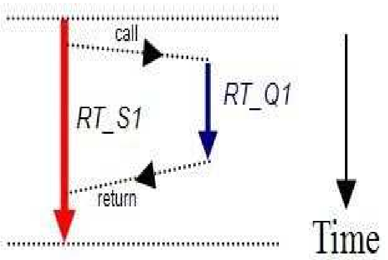


Fig. 4. The cumulative nature of the response time parameter

Thus, the response time metric can be primarily used in identifying entire faulty paths rather than individual faulty components in a dependency tree (e.g., the path T1—>S1—>Q1 instead of the component Q1). In many cases, this is not the adequate level of granularity for root cause identification. One key feature of our system is that it captures as the model variable the *local time* spent by transactions at a component, thus remaining independent of the response time of the children. Hence, a bottleneck at Q1 does not affect the behavior model of S1. The first step of our model-builder logic is to compute the local time spent by transactions at each component out of the cumulative response time data that is typically monitored. The algorithm for this computation is described in Section 3.C.

In the next step, the model-builder checks the current state of transactions. A transaction that generates a monitor violation notification means that its response time has been above the T_SLO limit and it is a "failed transaction". Otherwise, it is a "successful" transaction. Our management system uses the T_SLO violation notification together with the dependency graph to choose whether the computed local time is valid for the c_SLO computation. The rule states that if any (all) of the parent transaction(s) of a component failed (succeeded) when the local time is obtained, then the sample is (not) used for c_SLO threshold computation. In Fig. 3, for instance, S1 may access only Q1, or only Q2, or both, Q1 and Q2. When a local time value is obtained for Q2, the model-builder logic checks the current state of T1 and T2. If both T1 and T2 are successful, the threshold is updated with the new local time, as detailed in Section 3.C. If T1 or T2 exceeds (or approaches by less than a percentage δ_{SLO}) the T_SLO, a notification is received, and the problem determination logic is invoked.

The problem determination logic localizes the root causes of the potential failure. If c_SLO thresholds already exist for the components involved in a notified transaction, each component's current local time is compared to the corresponding c_SLO threshold. If the local time exceeds (or approaches by less than δ_{SLO}) the local threshold, the component is ranked as described in Section 4 and the root cause resource is identified.

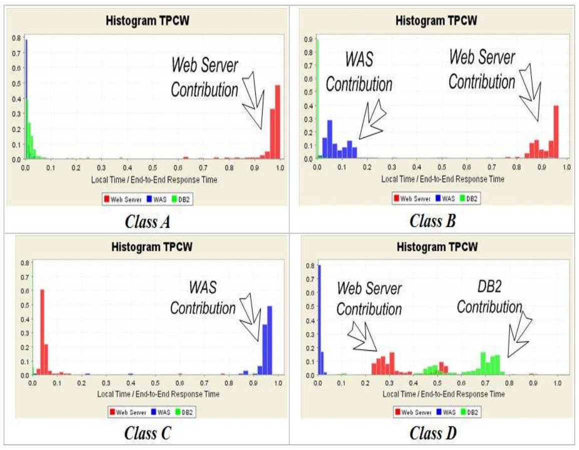


Fig. 5. Typical Transaction Behavior in E-business Applications

We have assumed up to this point that each component has a c_SLO threshold for each transaction in the system. For large applications and complex distributed environments, this may result in a large number of thresholds to manage. To address this inconvenience, we group transactions with similar behavior in a single class characterized by one threshold for each resource type involved in those transactions, thus reducing the total number of managed thresholds. Our technique allows comparing the behavior of different transactions over large periods of run-time and under various loads and requests conditions. Fig. 5 illustrates examples of behavior types encountered in a typical e-business application, modeled after TPC-W. They show the distribution of the component's local time contribution to the related transaction's end-to-end response time. We will refer to the results shown in Fig. 5 extensively in Section 5, where we experimentally investigate the categorization of transactions into behavior based classes.

C. Component_SLO Threshold Computation

The main idea of our technique is to combine system dependencies with the components' response time in order to translate the end-to-end user-defined T_SLO thresholds into individual dynamic c_SLOs. We first compute the time locally spent (T_{Local}) by specific transactions at each system component, for a stable load in the system. As expected, T_{Local} increases as the load increases, because the waiting time in the system increases. However, the T_{Local} fraction of the end-to-end transaction response time is independent of the load if the system components operate on the linear part of their response time curve [27], [28]. The servers involved in our scenario satisfy this assumption. The key observation in this approach is that the T_{Local} fraction of the end-to-end transaction response time reflects the individual c_SLO threshold fraction of the T_SLO.

Algorithm:

Let us consider the Web Server response time as RT_{WEB}. This response time at the Web Server includes the time (T_{Local_WEB}) spent by the request on the Web Server itself, as well as the response time (RT_{WAS}) of the next component involved in serving the transaction (in our case the Web Application Server). In a three level architecture with synchronous processes, as the one considered in our scenario, this is:

$$RT_{WEB} = T_{Local\_WEB} + RT_{WAS}$$

$$RT_{WAS} = T_{Local\_WAS} + RT_{DB2}$$

In general, the response time of component i:

$$RT_i = T_{Local\_i} + RT_{next\_resource}$$

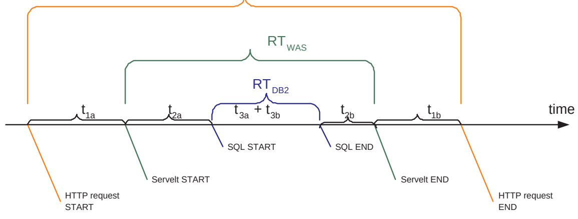


Fig. 6. Transaction response time decomposition in a three level e-business architecture

T_{Local} is the time taken for handling a request before the synchronous invocation of the next resource in the system, plus the time taken for handling and sending the response from that resource to the expected caller. In case of the Web Server, T_{Local_WEB} is illustrated in Fig. 6 by t_{1a}, the time before the Web Server invokes the corresponding servlet, and by t_{1b}, the time for handling and sending the servlet response to the user. In general:

$$T_{Local\_i} = t_{ia} + t_{ib}$$

T_{Local} is computed as the difference between the response times of successively invoked resources, for a known period of time and known load in the system. Let us consider transaction T1 from the dependency graph in Fig. 3, where servlet S1 invokes the collection S_SQL = {Q1, Q2}. Hence, S1's local time is the difference between S1's response time, $RT_{S1}(N)$, and the response time of S1_SQL triggered by S1, $RT_{S1\_SQL}(N)$:

$$T_{Local\_S1}(N) = t_{Sa} + t_{Sb} = RT_{S1}(N) - RT_{S1\_SQL}(N),$$

where N signifies the load in the system as measured by the number of customers issuing HTTP requests.

If there is only one SQL invoked by a servlet S in a transaction T, RT_{SQL} is computed as the average response time of that SQL. Otherwise, if the dependency graph shows that the servlet may trigger multiple SQLs (SQL_1, …, SQL_M), RT_{SQL} is computed as the average of the response times of these SQLs, weighted by their cumulative frequency of invocation during the period of time up to the current sample:

$$RT_{S\_SQL}(N) = \sum_{k=1}^{M} \frac{\#SQL_k}{\#S} \times RT_{SQL}(N),$$

where $\#S$ is the number of occurrences of servlet S and $\#SQL_k$ is the number of occurrences of SQL_k invoked by servlet S.

Hence, for a generic transaction T where resource$_i$ may invoke resource$_1$, …, resource$_K$,…, resource$_M$, the local time spent at resource$_i$ is computed as follows:

$$T_{Local_i}(N) = RT_i(N) - \sum_{k=1}^{M} \frac{\#_k}{\#_i} \times RT_k(N) \qquad (1)$$

Here we have assumed that the behavior of a component remains unchanged irrespective of whoever calls it in the same transaction context, and if all other parameters are the same. In future, our modeling logic can be easily extended to detect high variances in our component behavior models and automatically split into multiple sub-behavior models so that individual models do not have high variance.

Based on T_{Local} time obtained as described in Equation (1) the T_{Local} fraction of the end-to-end transaction T response time is computed as follows:

$$P_i = T_{Local_i}(N) / RT_{T\_end-to-end}(N) \qquad (2)$$

For successful transactions T, we store for each component the P_i values, which represent the history of the components behavior under transaction T. The comparison of transactions mentioned in the previous section is based on these P_i values. Changes in the environment may be reflected in these values.

The c_SLO for transaction T is computed based on the average P_i value:

$$c\_SLO_i = T\_SLO \times avg(P_i) \qquad (3)$$

where T_SLO is the transaction threshold (or other type of response time SLO limits, e.g., an interval).

As soon as a predetermined number of end-to-end T_SLO notifications have been received, the problem determination logic is invoked. At that moment, (i) the components invoked in the notified transaction(s) are identified based on the dependency graph, (ii) their T_{Local} is averaged over the period of time since the notifications have been received, and (iii) the average local time is compared to the correspondent c_SLO threshold. Finally, (iv) the affected components are ranked by severity degree as shown in the next section.

IV. Problem Determination

A. Severity Computation

The key component in the problem determination logic is the severity computation for each component. In our reactive PD approach, the severity value is defined as the measure of the component local time threshold violation during the current problematic period with respect to the c_SLO computed over the long term. The following steps are followed for severity computation.

```
1. for each component do
2. begin
3.     Severity = 0
4.     if $n$ $T_{Local}$ $>$ c_SLO //$n$=number of samples
5.     begin
6.         compute average bad $T_{Local}$
7.         severity = avg($T_{Local}$)/c_SLO
8.     end
9. end
```

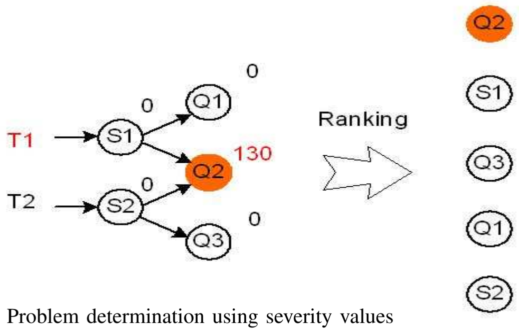


Fig. 7. Problem determination using severity values

Once the severity is computed, the components in each tier of the system are ranked by their severity value, as shown in Fig. 7. Due to our T_{Local} model parameter, the severity value of a parent is independent of problematic children and only the root-cause nodes are highlighted for the user in the resource dependency graph (Q2 in Fig. 7).

B. Problem Prediction

In this section, we describe our proactive techniques developed on the reactive problem determination algorithm, discussed so far. The techniques are used to derive relative ordering among the system components that indicates the most likely cause of an imminent problem.

The simplest technique to achieve proactiveness is to subdivide the end-to-end T_SLO and the internal c_SLO into multiple thresholds so that one can detect finer level changes or trends. The following steps are followed for severity computation, where n is number of samples.

```
1. for each component do
2. begin
3.         Severity = 0
4.         if $n$ $T_{Local}$ $>$ c_SLO - $\delta_{SLO}\times$ c_SLO
5.         begin
6.            compute average bad $T_{Local}$
7.            severity = avg($T_{Local}$)/c_SLO
8.         end
9. end
```

When the T_SLO monitor detects the end-to-end response time being close to the corresponding T_SLO beyond the safe limit of $\delta_{SLO}\times$ T_SLO, issues an imminent violation notification. As a result, each component related to the respective transaction starts comparing its T_{Local} to the proactive threshold, which is c_SLO - $\delta_{SLO}\times$ c_SLO. Here δ_{SLO} values may vary depending on whether intermediate prediction levels are required. Similar to the reactive PD approach, if T_{Local} exceeds the local threshold for a sufficient number n of samples, an average is computed from the T_{Local} obtained under problematic transaction(s) and subsequently used for computing the severity. Finally, the components in each system tier are ranked by their severity value and the top of the list predicts the most probable resources to imminently experience performance problems.

Note that in the proactive as well as in the reactive techniques, the c_SLO and T_SLO thresholds are computed as the means of their distributions. Higher moments of the distribution may also be considered for ranking components. In this case, we compute the severity value of a component as the normalized variance of the c_SLO distribution. The normalization is done by considering the percentage variance w.r.t. the mean of the distribution. A higher variance w.r.t. the mean is going to position a component higher in the suspect list. The ordered list may be computed continuously. Depending on the alert system implementation, the ordering based on higher moments may also be created when the end-to-end T_SLO threshold shows significant change. Thus, the end-to-end alerts are based on multiple split thresholds while internally normalized c_SLO variance is used for ordering suspect components on end-to-end notifications.

A second proactive technique uses the drift of the c_SLO for ordering components. In this case, the good model of a component is updated with response time samples that are less than T_SLO. However, the most recent k samples (called topK samples) are excluded from the computation of the good model, as they are used to compute a separate short-term good behavior model. The remaining of the samples are used to create a longer term good behavior model. The short-term and the long-term models are compared to detect drifts and create an ordering among components. The component with the highest drift is positioned on top of the suspect list. Our model is based on the mean of the set of samples considered. The drift is measured as the distance between the two means. The ordering may be continuously maintained or may be created when a proactive notification is received. Assuming that multiple thresholds are implemented, a notification is issued when an end-to-end response time crosses to a higher intermediate T_SLO threshold.

In case proactive T_SLO notification is not present, an internal proactive notification is generated when the short-term mean does not lie within m times the standard deviation of the long term mean. The two parameters mentioned above, m and k, are to be tuned. If we have high value of k, the mean of top_K model would be closer to its real value, however we shall be more reactive when a problem manifests itself in the system due to the large window size. On the other hand, if k is too small, we may detect false positives. Similarly, if m is too small we may generate false positives, while if it is too large we may be more reactive. This reasoning implies that m and k are inversely proportional. When a problem occurs in a component and its behavior deteriorates considerably from its normal behavior, i.e., the response times values from that component shooting up considerably from the normal responses expected from that component, then we should use a relatively higher value of m and smaller value of k. These settings also ensure a quick reaction time in case a problem occurs in the system. One of the down sides of this approach is that some times we may raise an alarm, whenever, there is spiky behavior in the system. This may be due to any number of reasons, for example, a spike in the workload arrival pattern may generate a spike at component level or a resource consuming process suddenly gets started at one of the servers which generate a spike at the component level at that the server. Nevertheless, these events like change in arrival rate or resource unavailability at a node are high level events and can be easily correlated with performance metrics.

The above mentioned techniques consider the load on the system to be fixed and are used to detect a failing component at constant arrival rate. Variations in the component behavior (e.g., response time increase) are systematically attributed to an abnormal operation of the component. However, in case of variable load in the system the component behavior variation can also be attributed to the changes in the arrival pattern or load. In order to provide proactive problem determination assistance under variable load, a complementary technique is discussed next. The core of our third proative technique is to build a model of the normal end-to-end response time (RT) as a function of the system load (λ) during an interval I chosen by the system administrator, based on trial runs, experience, etc, such as all the user transactions are exercised in the system a sufficient number of times to provide relevant statistics. The steps for building the RT function are described below. We assume the managed system is well provisioned to serve the maximum load allowed in the system is less than T_SLO. We also assume that the customer load is throttled to keep the input load in the zone of the linear part of the corresponding response time curve [27], [28].

1) During I if $RT <$ T_SLO read the transaction response time RT and correspondent system load λ.
2) Compute Pearson's correlation r between the load λ and the response time RT for each transaction type.
3) Compute the slope b and the 0y intercept A for the linear regression of λ and RT.
4) The estimated RT_i of transaction type i is computed as:

$$RT_i = b\lambda + A$$

Once trained, the prediction algorithm is enabled to proactively detect end-to-end application performance anomalies by comparing the system observed performance under variable load to the expected end-to-end performance computed by our algorithms, and trigger alarms:

1) If the monitored response time is smaller than the T_SLO and has a squared difference bigger than the maximum squared difference of the training data, the SLO monitor sends an early notification for failure prediction.
2) If the monitored response time is bigger than T_SLO, the SLO monitor sends an SLO violation notification for failure detection.

Upon reception of notification, the previously discussed techniques are applied to order the components. The resources on the top of the list are potential causes of the predicted failure.

V. Quality of PD using Inexact Dependency Graphs

We have described in the Section 3.C how the T_{Local} and c_SLO thresholds calculation relies on the resource dependency information. In case of ARM enabled systems, the dependency is accurately known from the monitored transaction traces. Therefore, the calculation of T_{Local} and c_SLO thresholds is exact. In case of legacy systems (not ARM instrumented) the dependency knowledge can be inferred using statistical approaches [2], [3]. Thus, false or missing dependencies may lead to an inexact dependency graph. In this section we analyze the impact of such flaws in the graph on the PD logic.

Equation (1) (Section 3.C) calculates the T_{Local} of a component i that may invoke resources *1*, ..., M, for a load N:

$$T_{Local\_i\_calc}(N) = RT_{i\_mon}(N) - \sum_{k=1}^{M} \frac{\#_k}{\#_i} \times RT_{k\_mon}(N)$$

We use the notations "mon" and "calc" to differentiate actual from calculated metrics. The contributors to the sum in the Equation above are the resources invoked by component i, as they are provided by the dependency graph. Let us suppose that among the M resources in the dependency graph that component i may invoke, there are false dependencies. Then:

$$T_{Local\_i\_calc}(N) = T_{Local\_i\_mon}(N) - Error \qquad (4)$$

If the $Error$ triggered by the false dependencies is so high that T_{Local_calc} has a negative value, then this is a straightforward indication of false dependencies highly ranked in the dependency graph and a need to refine the graph. No c_SLO is to be computed as valid in this case.

If T_{Local_calc} has positive values for all resources involved in a transaction, we compare their sum with the actual end-to-end transaction average response time. Although the sum will be smaller than the actual end-to-end transaction average response time ($Error$ is a positive value), we analyze the difference. If the difference is higher than a preset threshold δ, the dependency graph and the c_SLO values should be recomputed. If the difference is smaller than δ, we accept as valid the c_SLO calculated from those T_{Local_calc}.

In the case of missing dependencies, T_{Local_calc} is positive but higher than the correct values for the respective parent resources. The sum of all T_{Local_calc} involved in a transaction will be higher than the actual end-to-end transaction average response time. Again, we accept as valid the c_SLO calculated in Equation (4) only if that difference is smaller than δ.

Note that experiments performed in [2], [3] show that dependency graphs, as used in this paper, are obtained with a precision and accuracy (as defined in [2], [3]) that lead to very low, and hence acceptable, δ values.

VI. Experimental Evaluation

This section outlines configuration and performance considerations of our application performance behavior modeling and PD technique on the testbed shown in Fig. 8.

A. Experimental Environment

The TPC-W bookstore application is a benchmark suite that models the behavior of online web merchant sites [29]. The workload generation is performed using the Remote Browser Emulation package [29]. The number of simultaneous customers specifies the simulated load in an experiment run. The think-time of requests is a uniform distribution with 7 seconds average for all experiments. TPC-W has 14 transactions and its resources are 14 servlets, 46 SQLs, 10 tables, and a database of 10,000 books.

We instrumented the TPC-W bookstore application source in order to get accurate transaction, servlet, and SQL instance level monitoring data. The Online Mining Engine (OME) in Fig. 8 extracts statistical dependencies between these resources based on existent averaged monitored data. We utilized the Performance Monitoring Infrastructure (PMI) [30] provided by WAS and the Snapshot interface provided by DB2 [31] to sample the aggregate monitoring data. OME stores the statistical dependencies in the CIM repository where they are available for the PD application.

A T_SLO monitor intercepts the HTTP requests and responses at the workload generator level and classifies the transactions as "successful" or "failed" after comparison of their response time to the T_SLO thresholds. We have set the T_SLO thresholds at 8 seconds for all 14 transactions as research [32] has indicated that most people will give up on a site if it takes more than 8 seconds to download a page and, therefore, this is the maximum acceptable time for a request to complete in a real customer environment.

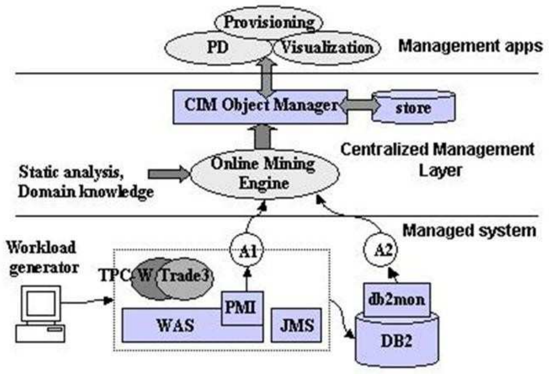


Fig. 8. Experimental set-up. WAS and DB2 are on 2GHz Pentiums with 1028 MB RAM

B. Generation of Behavior Classes

In the first part of our experiments, we collected T_{Local} for all the components of the 14 TPC-W transactions. The T_{Local} fraction of the end-to-end response time represents the P_i value computed as shown in Section 3.C. Fig. 5 illustrates 4 of the 14 distributions of the P_i values obtained for a load of 100 simultaneous customers using the instrumentation. We compared the distribution of the P_i among the 14 transactions and grouped similar ones in 6 classes as illustrated in Fig. 9. For example, six of the TPC-W transactions (Admin_Request, Customer Registration, Home, New_Products, Product_Detail, and Search_Request) have the behavior illustrated in the histogram Class A of Fig. 5. Other three transactions (Buy_Request, Execute_Search, and Shopping_Cart) have their P_i distributed as in the histogram Class B of Fig. 5. The lower-left histogram (Class C) characterizes another two transactions (Admin_Response and Order_Display). The lower-right histogram (Class D) corresponds to the Best_Sellers transaction. Finally, the two remaining transactions have their particular behavior that does not match the above four classes (Buy_Confirm in Class E and Order_Inquiry in Class F).

We repeated the transaction behavior comparison experiment with varying load in the system. The number of customers is increased till a significant percentage of URL requests time out due to high load. The WAS machine can support up to 150 customers on TPC-W in order for the system to operate on the linear part of its response time curve. We investigated the character of transaction classification, i.e., whether it is an application invariant or it is related to the load.

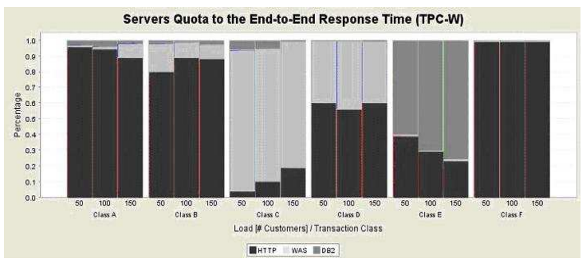


Fig. 9. Servers Quota of the End-to-End Response Time using accurate dependency and instrumented based data

Fig. 9 illustrates the results of this analysis. The T_{Local} distribution between the servers varies slightly with the load variation for a given class while the character of the class behavior is preserved. In order to test the impact of different traffic patterns, we ran additional experiments by varying the transaction mix for each load level. The results, not shown here, revealed the same behavior classification. Based on these observations, the model-builder logic computes the c_SLO thresholds for 6 classes of transactions instead of 14 transactions. However, if a major change takes place in the application (e.g., transaction addition/deletion) or in the computational environment, the behavior classification should be redone in order for the PD process to reflect the new context.

In the above experiments, precise dependency information was obtained through instrumentation. If instrumented code is not available, a statistical graph (with some false dependencies) may be obtained through the online mining techniques presented in [2], [3]. We took a TPC-W bookstore graph with 82% precision extracted at a load of 100 simultaneous customers. The inexactness reflected in the 82% precision (compared to the 100% precision of the true graph) corresponds to false servlet to SQL dependencies, in addition to the true ones.

We repeated the experiment using the statistical graph and aggregate monitoring data obtained from WAS and DB2 monitoring interfaces. As expected, the sum of the components' averaged P_i was slightly different from 100% due to the use of statistical data. Additionally, the DB2 average P_i is higher than in reality, while the servlet P_i is lower than in reality for the transactions altered by false dependencies (see Equation(2) in Section 3.C). The resource behavior inferred using statistical dependencies is very close to the accurate behavior model. In addition, the transaction categories yield the same composition. These results are not illustrated here due to lack of space.

However, dependency inexactness reflects on the inexactness of resource behavior models. Thus the quality of behavior modeling is limited by the quality of the dependency graph used. Our experimental results described in the next section investigate the impact of such inexactness and show that graphs with limited errors (such as 82% precision) can be successfully used for problem determination most of the time.

C. Problem Determination

In the second part of the experiments, we investigated the efficiency of our PD technique as well as the effect of the quality of the dependency graph on the PD results. A fault injector program inserts problems into the system by regularly locking chosen servlets on WAS, or tables on DB2, or both simultaneously. Thus, all transactions involving that particular servlet or table slow down, their response time increases over the T_SLO threshold, and the T_SLO monitor notifies their T_SLO violation. The injected problems are logged into a file as the ground truth and the file is used for evaluating the efficiency of our PD technique.

We evaluated the results of our experiments using two performance measures: *accuracy* and *precision*. *Accuracy* is defined as the percentage of the true faults discovered, averaged over multiple problem injections. $Precision$ is a performance measure in ranking the possible faulty components during PD, when a T_SLO violation is being investigated. We rank the possible faulty components in descending order of their severity value. Recall that the severity value of a component is the difference between its measured performance and the c_SLO. In the sorted list of problems numbered (starting from the highest severity value) from 1 to n, let $m(\leq n)$ be the last true faulty components in the list. We assigned a penalty of $m - i$ to the false problem labeled i, where $1 \leq i \leq m - 1$. The maximum possible penalty, when there are only false problems before the last true problem, is therefore $w_{tot} = m(m-1)/2$. Thus the total penalty due to false problems is the sum as follows:

$$w_f = \sum_{i<m,\ i\ is\ a\ false\ problem} m - i$$

We used the $percentage\ precision$ defined as $100(1 - \frac{w_f}{w_{tot}})$. Intuitively, this definition penalizes a false problem more if it occurs higher in the list. The closer the PD is to 100% $accuracy$ and $precision$, the higher is its efficiency.

We ran experiments with load values of 50, 100, 150 customers and three different transaction mixes over the duration of one hour each. During each experiment we randomly injected (1) only servlet problems, (2) only table problems, then (3) both servlet and table problems mixed. Each problem occurs 5 to 10 times and the average accuracy and precision are computed over the injected problems separately for each case (1), (2) and (3). We investigated the efficiency of our PD technique in the case of instrumented code, as well as the effect of false dependencies in our 82% precision dependency graph on the detection of servlet, table, and mixed problems. For the 82% precision dependency graph, the critical factor that impacts the PD performance is the low WAS P_i and high DB2 P_i values of the transactions altered by false servlet to SQL dependencies (see Equation (2) in Section 3.C). These values lead to low WAS and high DB2 thresholds and, thus, to false servlet problems that are ranked at comparable level with true table problems in the list of suspected root cause.

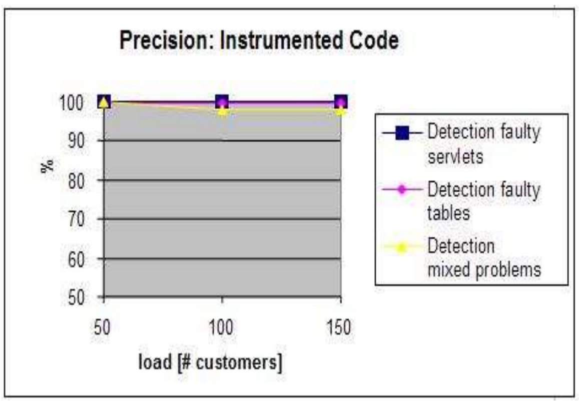


Fig. 10. Precision of the PD algorithm using exact dependency graph and instrumentation based monitoring data

We start by evaluating the accuracy of our technique. The results (not illustrated here) show that the accuracy is 100% in all situations. This means that we always find the injected problem in the ordered output list of suspected root causes. Fig. 10 and Fig. 11 show the variation of precision values with increasing customer load. As expected, the precision of our PD technique is higher when the code is instrumented compared to when we use aggregate monitoring data and a statistical graph. The root causes are almost always on the top of the suspected problem list when the dependency graph is exact (see the almost 100% precision in Fig. 10).

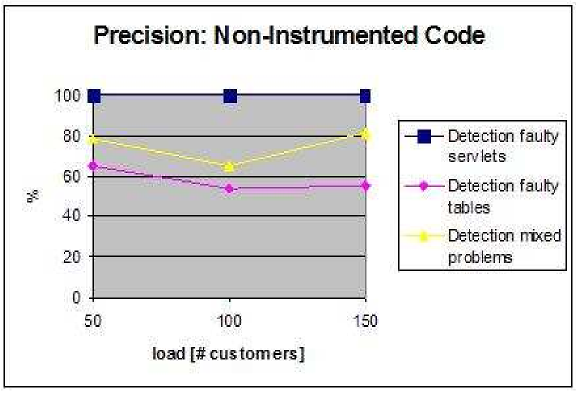


Fig. 11. Precision of the PD algorithm using statistical dependency graph and aggregate monitoring data

In case of statistical dependency, the root cause list may contain, as expected, few false problems highly ranked while detecting faulty tables among the real problems that are always identified and highly ranked (see the curves with diamonds and triangles in Fig. 11).

Note that the statistical dependency graph of 82% precision is extracted at a load of 100 simultaneous customers, which is a high load for our laboratory environment. In [2], the authors recommend the dependency extraction at low loads for highly precise dependency graphs. By using very precise dependency graph, our PD technique performs close to the results of Fig. 10, which were obtained using the instrumented code.

VII. CONCLUSIONS

In this paper, we have presented our research in the area of proactive problem determination in transaction oriented, multi-tier e-commerce systems, consisting of HTTP, applications, and database servers. The originality of our approach, as compared to others reported in the literature, is that we use a combination of resource dependency information and resource operational models to facilitate the rapid isolation of causes when application performance degrades. The PD system we have built uses a two-phased approach. In the first phase, user transactions are launched with varying rates to compute their dependencies on the resources that support them. In addition, important metrics related to the resources are monitored and analyzed to obtain a profile of their behavior. In the second phase, the user transactions are grouped into categories, based on the run-time profiles of their dependent resources, and appropriate thresholds are set.

Our validation experiments performed in the laboratory, in reasonable realistic setups, have shown that this approach yields a promising level of efficiency in root cause analysis. As future work, we plan to validate our results and refine our algorithms using different types of transactions, such as those that use messaging middleware.

REFERENCES

[1] A. Gillen, D. Kusnetzky, and S. McLaron, "The role of linux in reducing cost of enterprise computing," IDC white paper, 2002.

[2] M. Gupta, A. Neogi, M. Agarwal, and G. Kar, "Discovering dynamic dependencies in enterprise environments for problem determination," in *Proc. of IEEE International Workshop on Distributed Systems Operations and Management DSOM'03*, Heidelberg, Germany, 2003, pp. 221–2003.

[3] M. K. Agarwal, M. Gupta, G. Kar, A. Neogi, and A. Sailer, "Mining activity data for dynamic dependency discovery in e-business systems," *eTransactions on Network Service Management eTNSM Journal*, vol. 1, no. 2, 2004.

[4] (2006) Cim: Common information model. [online]. available:. [Online]. Available: http://www.dmtf.org/standards/standard_cim.php

[5] K. Appleby, J. Faik, G. Kar, A. Sailer, M. Agarwal, and A. Neogi, "Threshold management in transaction based e-commerce systems," in *Proc. of the Ninth IFIP/IEEE International Symposium on Integrated Network Management IM'05*, France, 2005.

[6] J. Aman, C. Eilert, D. Emmes, P. Yocom, and D. Dillenberger, "Adaptive algorithms for managing a distributed data processing workload," *IBM Systems Journal*, vol. 36, no. 2, pp. 242–283, February 1997.

[7] M. Goldszmidt, D. Palma, and B. Sabata, "On the quantification of e-business capacity," in *On the Quantification of e-Electronic Commerce EC'01*, Florida, USA, 2001, pp. 235–244.

[8] Y. Diao, J. Hellerstein, and S. Parekh, "Stochastic modeling of lotus notes with a queueing model," in *Computer Measurement Group International Conference CMG'01*, California, USA, 2001.

[9] Z. Liu, M. Squillante, and J. Wolf, "On maximizing service-level-agreement profits," in *ACM Conference on Electronic Commerce EC'01*, Florida, USA, 2001.

[10] D. Menasce, "Two-level iterative queuing modeling of software contention," in *10th IEEE International Symposium on Modeling, Analysis and Simulation of Computer and Telecommunications Systems MASCOTS'02*, Texas, USA, 2002.

[11] W. Cleveland, D. Lin, and D. Sun, "Ip packet generation: Statistical models for tcp start times based on connection rate superposition," in *Proc. of ACM SIGMETRICS'02*, Santa Clara, California, USA, 2000, pp. 166–177.

[12] H. Wang, D. Zhang, and K. Shin, "Detecting syn flooding attacks," in *Proc. of IEEE INFOCOM*, New York, USA, 2002.

[13] M. Thottan and C. Ji, "Anomaly detection in ip networks," in *IEEE Transactions on Signal Processing*, 2003, vol. 51, pp. 11–14.

[14] C. Hood and C. Ji, "Proactive network fault detection," in *Proc. of IEEE INFOCOM*, Japan, 1997.

[15] W. Leland, M. Taqqu, W. Willinger, and D. Wilson, "On the self-similar nature of ethernet traffic," in *IEEE/ACM Transactions Networking*, 1994, vol. 2.

[16] M. Basseville and I. Nikiforov, *Detection of Abrupt Changes: Theory and Application*. Englewood Cliffs: Prentice-Hall, 1993.

[17] S. Steinke, "Tools for managing application performance," *Network Magazine*, June 2003.

[18] (2006) Dmtf: Distributed management task force. [online]. available:. [Online]. Available: www.dmtf.org

[19] (2006) Arm: Application response measurement. [online]. available:. [Online]. Available: www.opengroup.org/management/arm.htm

[20] M. Steinder and A. S. Sethi, "The present and future of event correlation: A need for end-to-end service fault localization," in *Proc. of Fifth World Multiconference on Systemics, Cybernetics, and Informatics MCSI'01*, Orlando, FL, 2001, pp. 124–129.

[21] J. Gao, G. Kar, and P. Kermani, "Approaches to building self healing systems using dependency analysis," in *Proc. of IEEE/IFIP Network Operations and Management Symposium NOMS'04*, Seoul, Korea, 2004, pp. 119–131.

[22] A. Brown, G. Kar, and A. Keller, "An active approach to characterizing dynamic dependencies for problem determination in a distributed environment," in *Proc. of IEEE/IFIP International Symposium on Integrated Network Management IM'01*, Seattle, WA, 2001, pp. 377–390.

[23] U. Blumenthal, G. Kar, , and A. Keller, "Classification and computation of dependencies for distributed management," in *Proc. of IEEE Symposium on Computers and Communications ISCC'00*, Antibes Juan Les Pins, France, 2000, pp. 78–83.

[24] M. Steinder and A. Sethi, "Probabilistic event-driven fault diagnosis through incremental hypothesis updating," in *Proc. of IEEE/IFIP International Symposium on Integrated Network Management IM'03*, Colorado Springs, CO, 2003, pp. 635–648.

[25] S. Bagchi, G. Kar, and J. L. Hellerstein, "Dependency analysis in distributed systems using fault injection: Application to problem determination in an e-commerce environment," in *Proc. of the 12th IEEE/IFIP International Symposium on Distributed Systems: Operations and Management DSOM'01*, Nancy, France, 2001.

[26] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Pd in large, dynamic internet services," in *Proc. of the International Conference on Dependable Systems and Networks*, Washington DC, 2002, pp. 595–604.

[27] L. Slothouber, "A model of web server performance," in *Proc. of the 5th Conference World Wide Web WWW'96*, Paris, France, 1996.

[28] J. Steffan. (2001) Performance monitoring for db2 udb. [online]. available:. [Online]. Available: www.quest.com

[29] Tpcw wisconsin website. [online]. available:. [Online]. Available: http://www.ece.wisc.edu/~pharm/tpcw.shtml

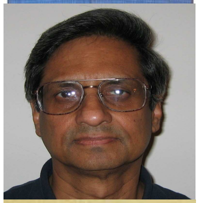
[30] S. Rangaswamy, R. Willenborg, and W. Qiao, "Writing a performance monitoring tool using websphere application servers performance monitoring infrastructure api," *In IBM WebSphere Developer Technical Journal*, vol. 5, no. 1, February 2002.

[31] Db2. [online]. available:. [Online]. Available: http://www-3.ibm.com/software/data/db2/

[32] Z. Research, "The need for speed," 1999.

Anindya Neogi received his Ph.D. and M.S. in Computer Science from State University of New York at Stony Brook in 2002 and 1999, respectively and a B.E. in Computer Science and Engineering from Jadavpur University, Kolkata in 1995. He has worked in Cadence Design Systems, Rether Networks, and is a Research Staff Member at IBM Research, New Delhi since 2002. He is interested in performance and failure management in large-scale systems.

Manoj K. Agarwal received a B.E. in Electronics and Communication Engineering from IIT, Roorkee in 1997 and M.S. degree (Electrical) from University of Texas at Austin in 2001. He was with Infosys Technologies Ltd. from 1997 to 1999. He joined IBM Research, New Delhi in 2001 as a Research Staff Member. His research interests are in distributed systems, computer networks and performance modeling of computer systems.



Gautam Kar received his Ph.D. in Computer and Information Science from the Ohio State University, Columbus, Ohio. He joined the IBM T.J. Watson Research Center in 1983, where, currently, he is the Manager of the Systems and Network management Department. His research interests include management of distributed systems and network services.

Ruchi Mahindru is pursuing her Ph.D. in Computer Science at New Jersey Institute of Technology. She received her M.S. and B.S. in Computer Science from Lehman College-*CUNY*. At present, she is an intern in Systems and Network management group at IBM T.J. Watson Research Center . Aside from performance analysis and problem detection, her interests include data mining, machine learning, and computational biology.

Anca Sailer received her engineering degree from *Politechnica* University of Bucharest, Romania, in 1995, and D.E.A. and Ph.D. in Computer Science from *Pierre et Marie Curie* University of Paris, France, in 1997 and 2000, respectively. She was a Research Member at the Networking Research Laboratory at Bell Labs, Lucent Technology, between 2001-2003. Currently a Research Staff Member at IBM T.J. Watson Research Center, her research interests include Autonomic Computing and Web Services architecture.