January 23, 2007

RT0700

Computer Science   10 pages

Research Report

Automated Performance Problem Determiantion by Observing Service Demand

Sei Kato, Toshiyuki Yamane and Takahide Noagayama

IBM Research, Tokyo Research Laboratory
IBM Japan, Ltd.
1623-14 Shimotsuruma, Yamato
Kanagawa 242-8502, Japan

Limited Distribution Notice

This report has been submitted for publication outside of IBM and will be probably copyrighted if accepted. It has been issued as a Research Report for early dissemination of its contents. In view of the expected transfer of copyright to an outside publisher, its distribution outside of IBM prior to publication should be limited to peer communications and specific requests. After outside publication, requests should be filled only by reprints or copies of the article legally obtained (for example, by payment of royalties).

Automated Performance Problem Determination by Observing Service Demand

Sei Kato, Toshiyuki Yamane and Takahide Nogayama
IBM Research, Tokyo Research Laboratory
1623-14 Shimotsuruma, Yamato-shi, Kanagawa-ken, 242-8502 Japan
{*seikato, tyamane, nogayama*}*@jp.ibm.com*

Abstract

Automated determination of performance problems in today's complex Information Technology (IT) systems is a crucial step in the development of Autonomic Computing. A new approach to detect and determine performance problems in a distributed computing system is proposed. The approach is based on a simple principle that the value of a particular performance metric, service demand, does not vary in time regardless of external environment changes, such as transaction mix ratio changes and intensity changes. Thus an abrupt change of the service demand value suggests an occurrence of a performance problem. Based on this idea, the authors developed a system which detects and localizes the machine causing a problem automatically by estimating and observing the service demand value using network traffic data flowing through the target IT system. The developed system is suitable for large scale distributed computing systems, since the system can monitor performance problems simply by connecting to the mirror port of the network switch that the machines of the target IT systems are connected to. Verification experiment results for our system in two environments, a test environment and a commercial production environment, show the effectiveness of the system in detection and determination of performance problems.

1 Introduction

Autonomic Computing is a fundamental concept in managing current highly complex Information Technology (IT) systems. To realize an Autonomic Computing system, the system should be aware of problems, determine the root causes of the problems and take appropriate actions to solve the problems by itself [6, 10, 9]. For performance problems, it still requires a lot of time and human efforts to detect and determine the causes of performance problems and the automation of performance problem determination has not been realized yet.

Current IT systems are composed of hundreds of heterogeneous components: network devices from multiple vendors, multiple operating systems, various vendor middleware and composite application software. In this distributed computing system, the root causes of performance problems can be hidden anywhere from hardware failures to middleware mis-configurations, from the client side to the back-end server side or in the other computing system working in concert with each other. Also, such systems are often accessed by the general public and therefore performance varies with individual user requests. Thus today's distributed computing systems make it difficult and time-consuming to detect and determine the causes of performance problems.

Over the past years, a considerable number of studies have been carried out on performance diagnosis systems. The most common approach employs artificial intelligence methods (see references in Hellerstein [4]). More recently, Hellerstein proposed a performance diagnosis system which explains the impact of specific causes of problems quantitatively by using a measurement navigation graph [4, 3]. While these approaches succeed in formulating the representation of knowledge about the system to be managed, no approach for actually using these ideas has yet emerged due to the difficulties in the acquisition of intensive knowledge. In terms of some components that organize IT systems such as database management systems and application server software, there exist performance diagnosis systems that are in commercial use [17, 11, 5, 2]. However, these rule-based performance advisers work well only in components where the architecture and behaviors are well known and do not work for entire IT systems. Thus there have not yet been any practical systems that can determine the root cause of the end-to-end performance problems of IT systems.

The reasons for the failures of the previous approaches lie in the lack of the universality of the knowledge and difficulties in the acquisition of knowledge. To overcome these difficulties, in this paper, we propose a model-based approach which requires a simple but universal minimum model. Our idea is based on the principle that the service demand, which represents the processing time for each service, do not vary over time. This principle allows us to conclude that if the service demand value changes for a service, some performance problem has appeared in the service. Thus we can detect performance problems and determine the root causes of the problems by observing the service demand for each service.

Based on this idea, we provide an automated performance problem determination system for practical use in the real world. The strengths of our approach are twofold. First, our model has universality and works for any IT system. Second, the model is simple and requires no prior research such as the parameter estimation of a queueing network model approach. These two advantage provides an easy implementation of our approach and help in practical use.

This paper is organized as follows. In Section 2, we describe our model to detect and determine performance problems. Section 3 presents an implementation of our model. We provide a system description and methods to measure the call counts and busy times. In Section 4 and in Section 5, we discuss the experimental results of our approach in a test environment and a production environment, respectively. Section 6 is devoted to our concluding remarks.

2 Model Formulation

Performance metrics such as response time and resource utilization vary according as changes of traffic intensity and of transaction mix ratios. The values of these metrics themselves would be meaningful for system administrators, but observing these values contributes little to performance problem detection and determination since their values change as the environment changes. Instead we adopt a performance metric, service demand, which is defined as a processing time for each service. This performance metric has the desirable property that the value is robust against intensity changes and transaction mix ratio changes. Thus, we can expect that if the value changes, some performance problem has occurred. In this section, we provide our model based on the service demand.

2.1 Problem Determination using Service Demand

Let us now consider a distributed computing system which consists of a set of multiple server machines $M = \{M_1, M_2, \dots, M_l\}$ and a set of multiple services $S = \{S_1, S_2, \dots, S_n\}$ which are running on server machines in the computing system. Here we use the term "service" as a service provided by server machines. Figure 1 shows a schematic view of the calling relationship of the services

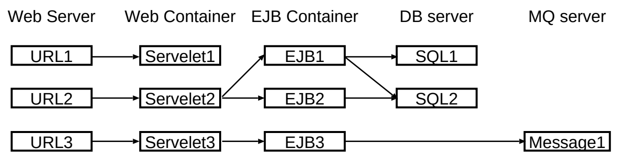


Figure 1. Calling relationship of services in a typical J2EE web system.

for a typical J2EE Web system, which is composed of 5 server machines: Web server, Web container, EJB container, database (DB) server and message queuing (MQ) server. In this Web system, the Web server serves content identified by a URL in responding to each client HTTP request. The Web Application server serves servlets/JSPs and EJBs in response to the requests from Web servers and Web containers. Similarly the DB server serves queries that are identified by SQL sentences, whereas the MQ server serves messages in response to EJB requests. Thus in this example, 12 services are running on 5 server machines, i.e., $l = 5$ and $n = 12$.

Now we define a performance metric, service demand d_{ik} as the processing time for service S_i at machine M_k. This value represents the processing time for a request for each service from a given machine resource. The value is of use for performance problem detection and determination in that the value is robust to environmental changes. Since this value does not contain any waiting time, the value is not affected by intensity changes. Also the value is robust to transaction mix ratio changes since the value is defined for each service. Thus the value is constant over time and is not affected by environmental changes.

It follows that an abrupt change of the service demand value indicates the occurrence of some performance problem. Thus it is expected that we can detect performance problems by observing this value. In addition, since this value is intrinsic to server machines, we can determine which machine is having a performance problem. In the example of Fig. 1, if the value of service demand for the services "SQL1" and "SQL2" changes, we can conclude that the performance problem occurred in the DB server machine. Thus we can isolate the machine causing a performance problem from among multiple server machines.

2.2 Estimation of Service Demand Value

As discussed in the previous subsection, we can detect and determine performance problems by observing the service demand values. However, since the values cannot be obtained explicitly, we have to estimate the values using observable performance metric values. The service demand can be estimated from busy times and call counts for each service.

Let us observe performance metrics from T until $T + \sum_{t=1}^{m} \Delta T_t$ with the observation interval ΔT_t. The observation interval ΔT_t can be different for each t as long as the observed values can be regarded as statistically steady. The observed values would be statistically unstable during the observation period if we take the observation interval ΔT_t as 24 hours, whereas the value would be steady in a statistical sense if we take the observation interval as 1 minute. Hereafter we refer to the period $\left[T + \sum_{t=1}^{j-1} \Delta T_t, \quad T + \sum_{t=1}^{j} \Delta T_t\right]$ as the observation period T_j and denote the value observed in this period with the suffix j.

Let b_{jk} denote the busy time of the machine M_k in the observation period T_j and a_{jik} denote the call counts for S_i called at the machine M_k in the observation period T_j. Then the utilization law [1, 13] can be expressed as

$$b_{jk} = \sum_i a_{jik} d_{ik}.$$

Here we define "busy time" as the time when the machine is busy during an observation period. The observed busy time is considered to include the observation error. Thus we introduce the following linear model

$$b_{jk} = \sum_i a_{jik} d_{ik} + \epsilon_{jk}, \tag{1}$$

where ϵ_{jk} denotes the observation error for the busy time b_{jk} at the machine M_k in the observation period T_j that satisfies the following conditions:

$$\langle \epsilon_{pq} \rangle = 0, \quad \langle \epsilon_{pq} \epsilon_{rq} \rangle = {\sigma_q}^2 \delta_{pr}, \quad \mathrm{N}\left(0, \sigma_q^2\right).$$

Here $\langle \cdot \rangle$ indicates the ensemble average of the metric $\cdot$, σ_q denotes the standard deviation of observation error at the machine M_q and δ_{pr} denotes the Kronecker delta.

For a fixed machine M_k, given a set of observational data, $\{b_1, b_2, \ldots, b_m\}$ and $\{a_{1i}, a_{2i}, \ldots, a_{mi}\} (1 \leq i \leq n)$, the best estimator for the service demand d_i is obtained by minimizing the error sum of squares $Q = \sum_{j=1}^{m} \epsilon_j^2$. The suffix k is omitted since we focus on a specific machine M_k. The best estimator $\hat{\boldsymbol{d}}$ is given as a solution of the normal equation

$$\boldsymbol{A}^t \cdot \boldsymbol{A} \hat{\boldsymbol{d}} = \boldsymbol{A}^t \boldsymbol{b}, \tag{2}$$

where $\boldsymbol{A} = (a_{ji}), \hat{\boldsymbol{d}} = (\hat{d}_1, \ldots, \hat{d}_n)^t, \boldsymbol{b} = (b_1, \ldots, b_m)^t$ and $\hat{\cdot}$ denotes the estimator of $\cdot$. Note that the information matrix $\boldsymbol{X} = \boldsymbol{A}^t \cdot \boldsymbol{A}$ of Eq. (2) can be decomposed to $\sum_{j=1}^{m} a_{ij} a_{ji} = \sum_{k=1}^{m-1} a_{ij} a_{ji} + a_{im} a_{mi}$. Similarly the right-hand side of Eq. (2) can be decomposed to $\sum_{j=1}^{m} a_{ij} b_j = \sum_{j=1}^{m-1} a_{ij} b_j + a_{im} b_m$. This tells us that we can update both the information matrix and the right-hand side of the normal equation incrementally and thus we can estimate the service demand online. Hence, service demand can be estimated with higher precision as we monitor longer times according to the law of large numbers.

Since service demand is a key parameter in performance models such as a queueing network model, this algorithm will also contribute to the performance parameter estimation of the system performance model. The relevance of the work by Liu *et al.* should be mentioned here. They propose an approach for parameter estimation in the queueing network framework using inference techniques [12]. It is interesting that in our case we can create a performance model without knowing the details of a system, since we can obtain both the calling relationships of each service and their service demands just by capturing the network traffic data.

We note here the rank of the design matrix $\boldsymbol{A}$. There are two possibilities when the design matrix is not of full rank, i.e., rank$(\boldsymbol{A}) =$ rank$(\boldsymbol{A}^t \boldsymbol{A}) < n$. The first is the case when the transaction mix ratio does not change in observation period T_j. That is, for all $p, q \in \mathbb{Z}$ $(1 \leq p < q \leq m)$, there exists $k \in \mathbb{R}$ such that

$$\hat{\boldsymbol{a}}_p = (a_{p1}, \ldots, a_{pn})^t = k \hat{\boldsymbol{a}}_q.$$

In this case, the rank of the design matrix is 1 and Eq. (2) becomes indeterminate. Although the transaction mix ratios do not change in a short monitoring period, the ratio is expected to change in a time scale like 24 hours. Thus we can avoid this situation by observing the data for a sufficiently long time. The second is the case when there exists a service with a linear dependence relationship with each other. For example, the number of times when the "login" service is called is expected to coincide with the number of times when the "logout" service is called. This problem can be avoided by combining these linearly dependent services into a new independent service.

2.3 Problem Detection by Residual Analysis

The point of the previous subsection is that we can estimate the service demand for each service and machine with accuracy by monitoring for a long time. This means, conversely, that it requires a long time to acquire the service demand value with high accuracy and therefore requires a long time to detect the performance problems at the service levels. Here, we will use the term "at the service levels" to refer to determining which service is having a performance problem and the term "at the machine level" to refer to determining which machine is having a performance problem. The best way to detect the performance problems with short detection delays is to abandon the problem determination at the service level and choose instead to determine the problem at the machine level.

Equation (1) holds regardless of intensity changes and transaction mix ratio changes. Therefore, if Eq. (1) is not true for a certain observation period T_j and for a specific machine M_k, this indicates that in the machine M_k, there exists a service whose service demand has changed. Thus, we can detect and determine performance problems at the machine level by checking the validity of Eq. (1). We can check the validity of the equation by using the following residual analysis. First estimate $\hat{d}_{ik}$ using the training run period data $b_{jk}, a_{jik}, (0 \le j \le m')$. Then an estimate of the error variance $\hat{\sigma}_k^2$ is obtained as $\hat{\sigma}_k^2 = \sum_{j=1}^{m'} (b_{jk} - \sum_i a_{jik}\hat{d}_{ik})^2/m'$. Next calculate the residual $r_{jk} = b_{jk} - \sum_i a_{jik}\hat{d}_{ik}$ using the data $\hat{d}_{ik}$ and the data $b_{jk}, a_{jik}(m' < j \le m)$ observed after the training period. The validity of Eq. (1) is checked by comparing the residual r_{jk} and the estimate of the error variance $\hat{\sigma}_k^2$ for each observation period T_j. Thus we can conclude that at a certain observation period T_j, the performance problem has occurred in the machine M_k if the absolute value of the residual r_{jk} exceeds a control limit such as 3 $\times\hat{\sigma}_k$ for example, i.e.,

$$|r_{jk}| > 3 \times \hat{\sigma}_k. \tag{3}$$

The geometric interpretation of Eq. (1) should be commented on here. Eq. (1) denotes that for a certain k, the observed values $(a_{j1k}, a_{j2k}, \ldots, a_{jnk}, b_{jk})$ plot on the hyperplane in $\mathbb{R}^{n+1}$ with the "thickness" of $\hat{\sigma}_k$ in a normal period. Eq. (3) denotes that in an abnormal time period, the observed values are off the "plate" of the hyperplane.

3 Implementation

Based on the ideas described in the previous section, we implemented our approach and developed a system which detects and determines performance problems using network traffic data. This network-based system is dominant over an agent system in that the system is suitable for large scale computing systems. Also our approach has the advantage that it has the smallest possible monitoring impact on a target computing system. In this section we describe the proposed system in detail.

3.1 System Description

Figure 2 shows a block diagram of the proposed system. The system outputs an event log after analyzing the input data, the network traffic data flowing through the target server machines. The system is connected to the mirror port of the network switch to which the target server machines $M_1, M_2, \cdots, M_l$ are connected and the packet monitor subcomponent captures the network traffic data which flows through the machines from the mirror port. Then the packet analyzer subcomponent measures a_{jik} and b_{jk}

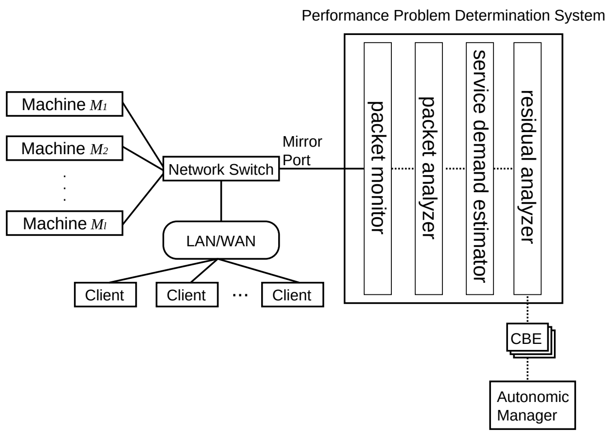


Figure 2. Block diagram of the proposed system. The dashed lines denote the network connectivity between machines and the dotted lines denote the flow of data. The system is composed of four subcomponents: packet monitor, packet analyzer, service demand estimator and residual analyzer.

by analyzing the captured network traffic data with the algorithms described in Subsection 3.2 and Subsection 3.3. Based on this data, the service demand estimator subcomponent estimates the service demand d_{ik} for each service S_i and machine M_k by solving the normal equation (2). For each observation period T_j, the residual analyzer subcomponent compares the residual r_{jk} with the estimate of the error variance $\hat{\sigma}_k$. When the calculated residual exceeds the control limit, the analyzer creates an event in the Common Base Event (CBE) format [8], which is a common log format based on the Web Services Distributed Management (WSDM) event format [15]. Finally the Autonomic Manager gives a diagnosis based on the CBE and proposes an appropriate remedy for the problem.

Here we briefly summarize the data flow of our system. The system captures packet data and measures call counts and busy times as described in the following subsections. Then the service demand values for each service as well as estimates of the errors are calculated by the least square method described in Subsection 2.2. Finally the residual analyzer subcomponent calculates the residual and detects performance problems by comparing the residuals and the estimates of the error variances.

3.2 Measurement of Call Counts

As seen in the previous subsection, the proposed system estimates service demand values using both call counts and busy times, which are measured from network packet data. In this subsection we see how the packet analyzer subcomponent measures call counts from network packet data.

In a distributed computing system, user requests are processed as a parent transaction on a server machine calls a child transaction on another server machine. Here we refer the term “transaction” as an instantiation of each service. The schematic view of a sample transaction flow is shown in Fig. 3. The server machine M_2 receives a processing request for the transaction X_2 from a parent transaction X_1 on the server machine M_1 at time $T_{receive,2}$ and then the machine M_2 starts processing the transaction X_2 or the transaction is queued. After processing, the transaction sends a processing request for the child transaction X_3 to the server machine M_3 at time $T_{receive,3}$. After the transaction X_3 has been processed at the machine M_3, the server machine M_2 receives the reply message for the transaction X_3 at time $T_{send,3}$ and then sends a reply message for X_2 to machine M_1 at time $T_{send,2}$. Note that the network delay is negligible since the messages are transmitted to machines connected to the same network switch. By capturing and analyzing the packet data on these transactions, we can obtain the following four data for each transaction $X_p (1 \leq p \leq s)$:

- request message receive time $T_{receive,p}$,
- request message send time $T_{send,p}$,
- the service $S(p)$ to which the transaction X_p belongs and
- the server machine $M(p)$ at which the transaction X_p is processed.

With this definition, the call count for each service during the observation period T_j is evaluated as

$$\begin{aligned} a_{jik} &= \#\Big\{p|S(p) = S_i, \quad M(p) = M_k, \\ &\qquad T + \sum_{t=1}^{j-1} \Delta T_t \leq T_{receive,p} < T + \sum_{t=1}^{j} \Delta T_t, \\ &\qquad T + \sum_{t=1}^{j-1} \Delta T_t \leq T_{send,p} < T + \sum_{t=1}^{j} \Delta T_t \Big\}. \end{aligned}$$

Thus the packet analyzer subcomponent measures call counts for each service S_i and for each machine M_k by counting the transactions that are processed during the observation period T_j.

3.3 Measurement of Busy Time

Besides the call counts a_{jik}, the packet analyzer subcomponent measures the busy time b_{jk} of each machine M_k in an observation period T_j. In this subsection, we describe how the subcomponent measures busy times using network traffic data.

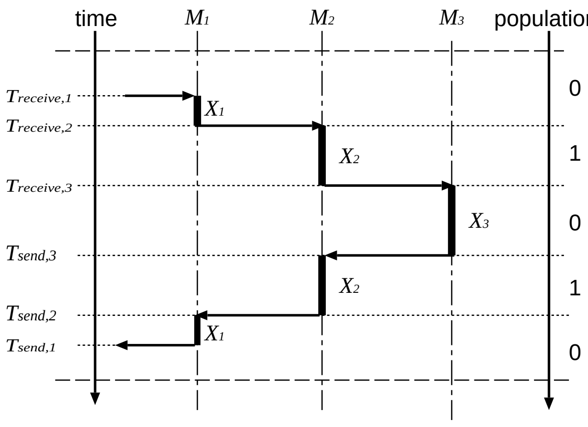


Figure 3. Schematic view of a sample transaction flow. The transaction X_1 calls the child transaction X_2 and the transaction X_3 is called from the parent transaction X_2. The heavy lines denote the periods when each machine is busy. The time evolution of the customer population of the machine M_2 is shown on the right vertical axis.

The server machine is expected to be busy if at least one transaction is being processed on the server machine. Thus we can measure the busy time b_{jk} in a machine M_k in an observation period T_j by summing up the periods when at least one transaction remains in the machine. We can examine these conditions by examining the customer population of each machine. Customer population in a sample transaction flow is shown on the right of Fig. 3. In this example, the customer population of the machine M_2 increases to 1 when the machine receives s processing request for transaction X_2 from the machine M_1 and decreases to 0 when the server machine M_2 sends the processing request of transaction X_3 to machine M_3. Likewise, the customer population of the machine M_2 increases to 1 when the machine receives the processing completion for transaction X_3 from the machine M_3 and decreases to 0 when the server machine M_2 sends the processing completion of transaction X_2 to machine M_1. The busy time of the server machine M_2 in this example is calculated as $T_{receive,3} - T_{receive,2} + T_{send,2} - T_{send,3}$. Thus busy time is measured by summing up the periods when the customer population is one ore more. The pseudo-code for this algorithm is shown in Algorithm 3.1.

Note that this algorithm works even in a multi-thread environment and regardless of such service disciplines as processor sharing (PS) and first-in-first-out (FIFO). The advantage of the method is that we can obtain busy times from network traffic data even when there exist multiple transactions in process and even when we do not know in detail about the service disciplines of the computer resources.

Some exception handling will be needed when we use this algorithm in a real environment. The customer population could be negative when a transaction sends multiple processing requests for child transactions and when an orphan transaction that has no parent transaction is called spontaneously. In the transaction fork case, the request completed message should not be sent until all replies of the forked transactions are received. Therefore, in this case, we can address the situation by allowing negative values for the customer population. The orphan transaction case occurs when an administrative server sends a heartbeat messages to cluster machines that are controlled and monitored by the administrative server. Our system estimates the calling relationships between transactions and therefore can detect these orphan transactions as those without parent transactions. Thus we can solve this problem by not counting these transactions in the customer population.

Algorithm 3.1: BusyTime($PacketData$)

```
comment: $N$: customer population
comment: $B$: busy time
comment: $T$: time stamp
procedure Diff($a, b$)
  return $(b - a)$

main
  $N \leftarrow 0$
  $B \leftarrow 0.0$
  while in an observation period
    do {
      if is received a processing request
      from the parent transaction
        then $N \leftarrow N + 1$
        else if is sent a processing completion
      to the parent transaction
        then $N \leftarrow N - 1$
        else if is sent a processing request
      to the child transaction
        then $N \leftarrow N - 1$
        else if is received a processing completion
      from the child transaction
        then $N \leftarrow N + 1$
        else do nothing

      if $N$ increases from 0 to 1
        then $T \leftarrow$ current time
        else if $N$ decreases from 1 to 0
        then $B \leftarrow B +$ Diff($T$, current time)
        else do nothing
  return $(B)$
```

4 Experimental Results in a Test Environment

We examined the effectiveness of our system in two environments: a test environment and a real production environment at a commercial site. In Section 4 and in Section 5, we discuss the experimental results in these two environments.

4.1 Experimental Settings

To verify whether the system can correctly detect a performance problem and localize the machine causing the problem, we carried out fault injection tests.

The machine configuration for the test environment is shown in Fig. 4. The system for the test consists of five machines, which are the workload generator, Web server, Web application server, database server and performance problem determination system. The target IT system is a typical three-tier Web system which consists of Web server, Web application server and database server. The online stock brokerage application named Trade3 [7], which is a sample benchmark application for the IBM WebSphere Application Server, is running on the application server. The workload is generated by creating pseudo-client requests using Web Performance Tools [14]. The services for the Web server are '/trade/app?action=login', '/trade/app?action=quote' and so forth. The services for the Web application server are '/trade/app?action=buy', '/trade/register.jsp' etc. Those for database server are such SQL sentences as 'select * from DB2INST1.CUSTOMER where CUSTOMERID =?' and 'select q1."PRICE",q1."COMPANYNAME" from QUOTEEJB q1' (abbreviated) and so on.

The machines for the target Web system and the workload generator are the IBM eServer xSeries 330 with 1 GB main memory and 1.4 GHz dual CPUs (Intel Pentium III) running Linux (RedHat Linux V8.0) and the machine for performance problem determination system is an IBM IntelliStation M Pro with 500 MB main memory and a 2.0 GHz CPU (Intel Pentium 4) running Linux (CentOS 4.3). Three of the four Linux machines run a Web server (IBM HTTP Server V1.3), a Web application server (IBM WebSphere Application Server V5.0) and a database server (IBM DB2 Universal Database V8.1), respectively, and these constitute the test Web system.

The machine for the performance problem determination system is connected to the mirror port of a Cisco Catalyst 4006 switch and copies packet data flowing through the four machines. The packet data is dumped using the tcpdump command and analyzed to obtain receive and send times for each transaction, the service to which the transaction belongs and the destination machine of the transaction. The performance problem determination system then calculates the busy times and call counts based on this data. Finally

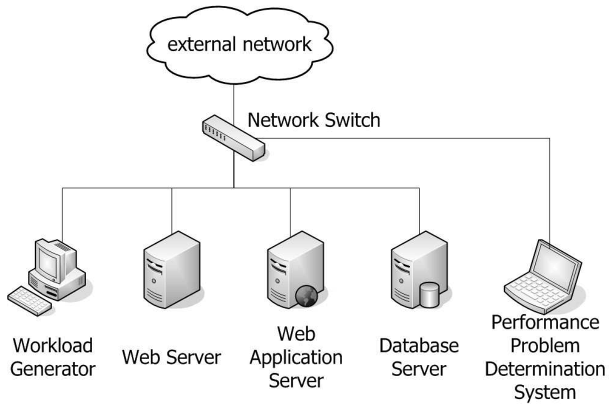


Figure 4. Test environment configuration. A workload generator and three servers are connected to a network switch. The proposed performance problem determination system is connected to the mirror port of the switch.

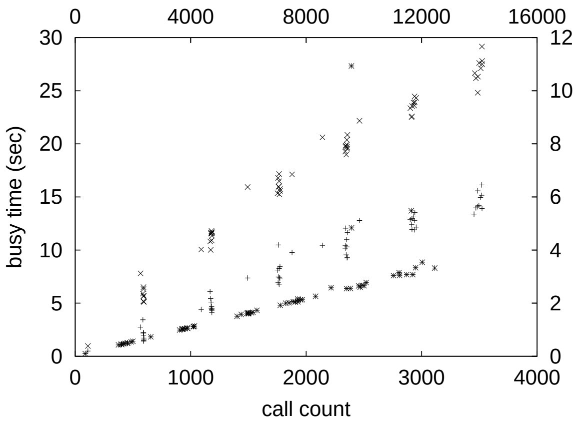


Figure 5. The relationship between busy times and call counts for the Web server (+), Web application server ($\times$) and database server ($*$). Data points (a_{jkk}, b_{jk}) for each server machine M_k ($k = 1, 2, 3$) and for each observation period T_j while varying the workload are plotted. The top and bottom horizontal axes show the call counts for the Web server (top), Web application server (top) and database server (bottom). The left and right vertical axes show the busy times for the Web server (left), Web application server (left) and database server (right).

performance problems are detected and determined by the residual analysis.

In this experiment, the workload is generated by requesting a fixed page transition scenario. Therefore, the transaction mix ratio is constant over time. We redefine the services which are running on the Web server as a single service. Similarly, we redefine the services on the Web application and database servers as a single service. Thus in this experiment, $l = 3$ and $n = 3$; each three service S_1, S_2 and S_3 is running on the Web server M_1, the Web application server M_2 and the database server M_3, respectively.

In Fig. 5, we plot the points (a_{jkk}, b_{jk}) for each server machine M_k ($k = 1, 2, 3$) and for each observation period T_j while varying the workloads. Here we use the notation a_{jkk} instead of a_{jik} since only one service is running on each server machine. In this experiment, the number of clients increases from 10 to 60 in increments of 10. The think time for the clients is fixed at 1 second. Although there exist some outliers in the busy time data of the database server, which occur because of packet loss, the busy time for each machine is proportional to the call counts. Thus we can see that the linear model works regardless of workload changes.

4.2 Fault Injection Test Results

To examine the ability of our system to detect and to determine performance problems, the following two fault injection tests were done. In both tests, the workload is steady over time with the fixed page transaction scenario.

The first fault injection test is to delete the database indexes online 16 minutes after the start of monitoring. A database index is defined for each table object to access the table data rapidly. The creation of a database index eliminates the need for full queries of all of the data stored in the table objects. Thus deleting the database indexes decreases search time performance.

The second test is to decrease the database buffer pool size from 2,000 to 50 pages online 10 minutes after the start of monitoring. A buffer pool is a finite area in memory that DB2 uses as a caching area for data for transactional processing, and for reading and writing data to and from disk. DB2 improves performance by minimizing the number of times a database has to retrieve data from disk. The size of DB2 buffer pool can be changed online and the decrease of the buffer pool size causes a performance degradation.

These experimental results are shown in Fig. 6. To detect the change points, we took 3-sigma as a control limit of a Shewhart control chart [16]. In the figures, the horizontal lines denote the upper control limit (UCL) for each server machine. The estimate of the error variance is calculated by using data $0 \leq j \leq 10$ for the first test and $0 \leq j \leq 5$ for the second test, respectively. The robustness of the change point detection is assured by issuing alarms only when the residual values exceed the control limit three times in a row. Here the observation interval ΔT_j equals one minute and therefore the alarm delay is three minutes.

We see that, in both tests, only the residual of the database server exceeds the UCL after the fault injection, whereas the others stay within the UCL. These results agree with our expectation in that only the residual of the database server into which the fault was injected exceeds the UCL. Thus the system localizes the machine with the performance problem correctly. After detection, this system triggers an alarm that a performance problem has appeared in the database server. These results show that the system can correctly detect the performance problem with a short delay and correctly localize the machine in which the performance problem has occurred.

5 Experimental Results in a Production Environment

To show the effectiveness of our system in a real production environment, our system was installed in a commercial system and monitored for performance problems for a week. In this section, we describe the health check test results in the production environment.

5.1 Test Configuration

The target system consists of five application servers and one database server. Our performance problem determination system, which is running on a ThinkPad R51 was connected to a network switch of the target system and captures the packet data flowing through the target Web system for one week. The observation interval ΔT_j was five minutes and thus we obtained about 2,000 data samples. On each Web application server, about 80 services were running.

5.2 Health Check Test Results

There were no performance problems in the monitoring period of one week. Now we show that our system successfully reported on the soundness of the monitored system. Our approach has two steps: first, the system estimates the service demand values and estimates of the error variances using the captured packet data during a training period; and second, it monitors for performance problems by comparing the residuals and the estimates of the error variances.

First, the system estimated the service demand values and the estimates of the error variances using the data captured in the training period. We used the first day as the training period during the week. In Fig. 7, we display

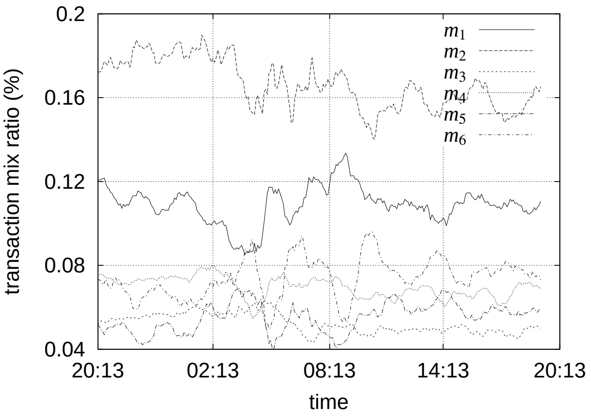


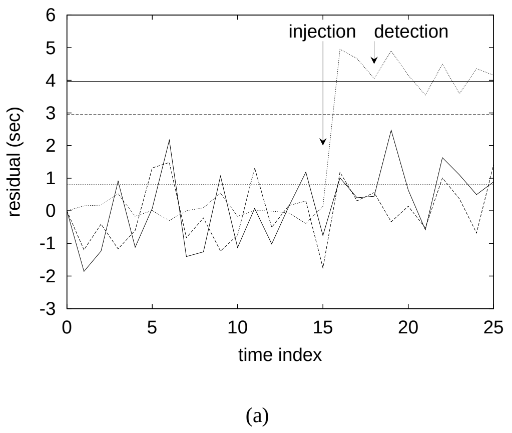
Figure 8. Time evolution of transaction mix ratios for six major services running on the machine M_1 during the training period. One-hour moving average lines are plotted.

the time evolution of the busy time b_{jk} and the total call count c_{jk} for each of the three Web application servers $M_k(k = 1, 2, 3)$ in the training period. Here c_{jk} denotes the sum of the call counts for the services, i.e., $c_{jk} = \sum_i a_{jik}$. The three Web application servers are equivalent in specification and the client requests are dispatched to the five Web application servers in round-robin order. As expected, we see that the busy times and call counts for each machine tend to have same values, although there are some outliers in the busy time data for the machine M_3. Fig. 7 also reveals that the call counts and busy times have a proportional relationship. This result suggests that Eq. (2) holds in this system.

To see the behavior in more detail, we show the time evolution of the transaction mix ratios in Fig. 8. In the figure, the lines show the one-hour moving averages for six major services running on a Web application server. The transaction mix ratio m_{jik} for the service S_i is calculated as $m_{jik} = a_{jik}/c_{jk}$. In accord with our expectations, we find that the transaction mix ratios for each service vary to some extent over time. The maximal variation in the one-hour moving average is up to 5.0 points.

The result of the transaction mix ratio changes over time enables us to estimate the service demand for each service using the least square fit method described in Subsection 2.2. Now we can calculate the service demand values for each of the six services and for each of the three machines using the call counts and busy time data from the training period.

Second, the system monitored for the performance problems in the monitoring period by comparing the residual and the estimated values during the training period. In Fig. 9, the time evolution of residual for each server in the monitor-



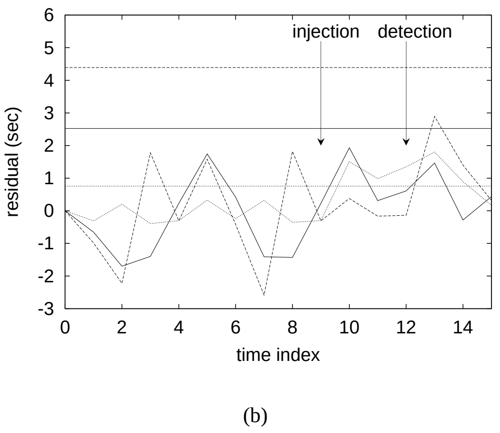
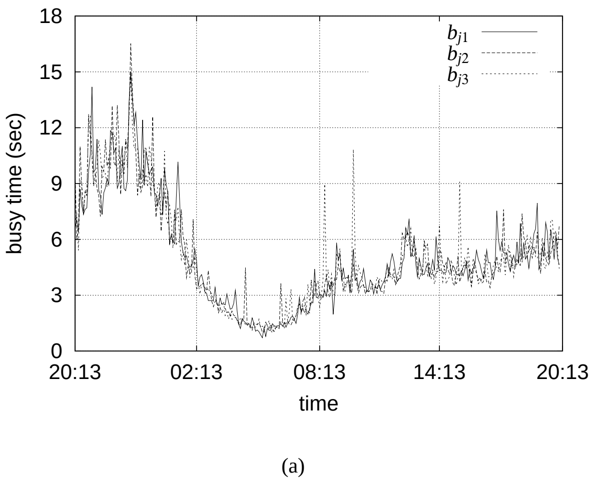


Figure 6. Time series graph of residuals for each fault injection test: (a) the index of the database is deleted at time $j = 15$; (b) the buffer pool sized is decreased at time $j = 9$. The solid line denotes values for the Web server, the broken line is for values of the Web application server and the dotted line is for values of the database server. The horizontal lines denote corresponding upper control limits.



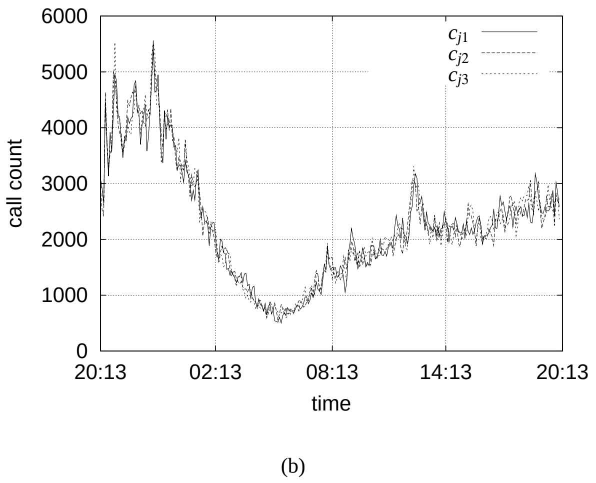


Figure 7. Time evolution of busy time (a) and call count (b) for each three web application server M_1, M_2 and M_3 in the production environment during the training period.

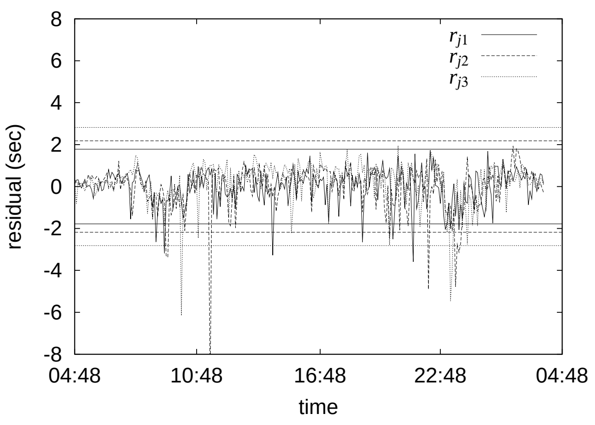


Figure 9. Time evolution of the residuals for each of the three server machines in the monitoring period. The corresponding horizontal lines denote the control limits for each server.

ing period is displayed. The corresponding horizontal lines denote the upper and lower control limits, which are taken as three times the estimate of the error variance. Although some data points exceed the lower control limit, there are no points which exceed the limit three times in a row and the system reports no errors. Thus we see the system successfully reports the soundness of the target commercial system.

6 Conclusion

In conclusion, we propose a new approach to detect and determine performance problems in a distributed computing system. This approach is based on the principle that an abrupt change of the service demand value suggests the appearance of performance problem. Therefore, observing the values of service demand enables us to detect and determine any performance problems. We developed an automated performance problem determination system that localizes the machine with a performance problem by capturing the network traffic data which flows through the target distributed computing system. Test results in two environments, a test environment and a production environment, show the effectiveness of the system in detection and determination of performance problems. Fault injection tests in the test environment test show that the system detects performance problem with only a short delay and correctly localizes the machine of cause. A health check test in a production environment succeeded in reporting on the soundness of that system. Verification experiments in a more complex computing system are now underway and those results will be reported elsewhere.

The authors would like to thank Hiroaki Etoh for his valuable comments. The first author is also grateful to the members of the IBM System Technical Service Center for their support.

References

[1] P. J. Denning and J. P. Buzen. The operational analysis of queueing network models. *ACM Computing Surveys*, 10(3):225–261, September 1978.

[2] R. H. High Jr. and M. Kloppmann. WebSphere programming model and architecture. *Datenbank Spektrum*, 8:18–31, Feburary 2004.

[3] J. L. Hellerstein. A comparison of techniques for diagnosing performance problems in information systems. *SIGMETRICS*, 1994.

[4] J. L. Hellerstein. A general-purpose algorithm for quantitative diagnosis of performance problems. *Journal of Network and Systems Management*, 11(1):199–216, March 2003.

[5] E. N. Herness, R. H. High Jr., and J. R. McGee. WebSphere application server: A foundation for on demand computing. *IBM Systems Journal*, 43(2):213–237, 2004.

[6] P. Horn. *Autonomic Computing: IBM's Perspective on the State of Information Technology*, 2001.

[7] International Business Machines Corporation. *WebSphere Application Server*. http://www-306.ibm.com/software/webservers/appserv/benchmark3.html.

[8] International Business Machines Corporation. *Best Practices for the Common Base Event and Common Event Infrastructure*, 2006. ftp://www6.software.ibm.com/software/developer/library/autonomic/books/cbepractice/index.pdf.

[9] S. Kato, J. Shimizu, T. Hama, T. Fukuda, and T. Yamane. Self-optimizing Web system. In *IEEE International Conference on Industrial Informatics (INDIN) 2003, Workshop on Autonomic Computing Principles and Architectures*, 2003.

[10] J. O. Kephart and D. M. Chess. The vision of autonomic computing. *Computer*, 36(1):41–50, 2003.

[11] S. S. Lightstone, G. Lohman, and D. Zilio. Toward autonomic computing with DB2 universal database. *ACM SIGMOD Record*, 31(3):55–61, 2002.

[12] Z. Liu, L. Wynter, C. H. Xia, and F. Zhang. Parameter inference of queueing models for IT systems using end-to-end measurements. *Performance Evaluation*, 63(1):36–60, January 2006.

[13] D. A. Menasce, L. W. Dowdy, and V. A. Almeida. *Performance by Design: Computer Capacity Planning by Example*. Prentice Hall PTR, 2004.

[14] R. Nesbitt. *alphaWorks : Web Performance Tools: Overview* . http://www.alphaworks.ibm.com/tech/wptools.

[15] Organization for the Advancement of Structured Information Standards (OASIS). *Web Services Distributed Management: Management Using Web Services (MUWS 1.1)*, 2006. http://docs.oasis-open.org/wsdm/wsdm-muws1-1.1-spec-os-01.htm.

[16] W. A. Shewhart. *Statistical Method From the Viewpoint of Quality Control*. Dover Publications, 1939.

[17] R. Telford, R. W. Horman, S. Lightstone, N. M. S. O'Connell, and G. M. Lohman. Usability and design considerations for an autonomic relational database management system. *IBM Systems Journal*, 2003.