Anomaly? Application Change? or Workload Change?

*Towards Automated Detection of Application Performance Anomaly and Change**

Ludmila Cherkasova[1], Kivanc Ozonat[1], Ningfang Mi[2] Julie Symons[1], Evgenia Smirni[2]
[1] HPLabs, Palo Alto & [2] College of William and Mary, Williamsburg
lucy.cherkasova@hp.com, kivanc.ozonat@hp.com, ningfang@cs.wm.edu, julie.symons@hp.com, esmirni@cs.wm.edu

Abstract

Automated tools for understanding application behavior and its changes during the application life-cycle are essential for many performance analysis and debugging tasks. Application performance issues have an immediate impact on customer experience and satisfaction. A sudden slowdown of enterprise-wide application can effect a large population of customers, lead to delayed projects and ultimately can result in company financial loss. We believe that online performance modeling should be a part of routine application monitoring. Early, informative warnings on significant changes in application performance should help service providers to timely identify and prevent performance problems and their negative impact on the service. We propose a novel framework for automated anomaly detection and application change analysis. It is based on integration of two complementary techniques: i) a regression-based transaction model that reflects a resource consumption model of the application, and ii) an application performance signature that provides a compact model of run-time behavior of the application. The proposed integrated framework provides a simple and powerful solution for anomaly detection and analysis of essential performance changes in application behavior. An additional benefit of the proposed approach is its simplicity: it is not intrusive and is based on monitoring data that is typically available in enterprise production environments.

1 Introduction

Today's IT and Services departments are faced with the difficult task of ensuring that enterprise business-critical applications are always available and provide adequate performance. As the complexity of IT systems increases, performance management becomes the largest and most difficult expense to control. We address the *problem* of efficiently diagnosing essential performance changes in application behavior in order to provide timely feedback to application designers and service providers. Typically, preliminary performance profiling of an application is done by using synthetic workloads or benchmarks which are created to reflect a "typical application behavior" for "typical client transactions". While such performance profiling can be useful at the initial stages of design and development of a future system, it may not be adequate for analysis of performance issues and observed application behavior in existing production systems. For one thing, an existing production system can experience a very different workload compared to the one that has been used in its testing environment. Secondly, frequent software releases and application updates make it difficult and challenging to perform a thorough and detailed performance evaluation of an updated application. When poorly performing code slips into production and an application responds slowly, the organization inevitably looses productivity and experiences increased operating costs.

Automated tools for understanding application behavior and its changes during the application life-cycle are essential for many performance analysis and debugging tasks. Yet, such tools are not readily available to application designers and service providers. The traditional *reactive* approach is to set thresholds for observed performance metrics and raise alarms when these thresholds are violated. This approach is not adequate for understanding the performance changes between application updates. Instead, a *pro-active* approach that is based on *continuous* application performance evaluation may assist enterprises in reducing loss of productivity by time-consuming diagnosis of essential performance changes in application performance.

With complexity of systems increasing and customer requirements for QoS growing, the research challenge is to design an *integrated framework of measurement and system modeling techniques* to support performance analysis of complex enterprise systems. Our goal is to design a framework that enables automated detection of application performance changes and provides useful classification of the possible root causes. There are a few causes that we aim to detect and classify:

- *Performance anomaly.* By performance anomaly we mean that the observed application behavior (e.g., current CPU utilization) can not be explained by the observed application workload (e.g., the type and volume of transactions processed by the application suggests a different level of CPU utilization). Typically, it might point to either some unrelated resource-intensive process that consumes system resources or some unexpected application behavior caused by not-fully debugged application code.
- *Application transaction performance change.* By transaction performance change we mean an essential change (increase or decrease) in transaction processing time, e.g., as a result of the latest application update. If the detected change indicates an increase of the transaction processing time then an alarm is raised to assess the amount of additional resources needed and provides the feedback to application designers on the detected change (e.g., is this change acceptable or expected?).

It is also important to distinguish between *performance anomaly* and *workload change*. A performance anomaly is indicative of abnormal situation that needs to be investigated and

*This work was completed in summer 2007 during N. Mis internship at HPLabs. E. Smirni is partially supported by NSF grants ITR-0428330 and CNS-0720699, and a gift from HPLabs.

resolved. On the contrary, a workload change (i.e., variations in transaction mix and load) is typical for web-based applications. Therefore, it is highly desirable to avoid false alarms raised by the algorithm due to workload changes, though information on observed workload changes can be made available to the service provider.

The rest of the paper is organized as follows. Section 2 introduces client vs server transactions. Section 3 provides two motivating examples. Section 4 and Section 5 introduce two complementary techniques as an integrated solution for anomaly detection and application performance change. Section 6 presents a case study to validate the proposed techniques. Section 7 describes related work. Finally, a summary and conclusions are given in Section 8.

2 Client vs Server Transactions

The term *transaction* is often used with different meanings. In our work, we distinguish between a *client transaction* and a *server transaction*.

A client communicates with a web service (deployed as a multi-tier application) via a web interface, where the unit of activity at the client-side corresponds to a download of a web page. In general, a web page is composed of an HTML file and several embedded objects such as images. This composite web page is called a *client transaction*.

Typically, the main HTML file is built via dynamic content generation (e.g., using Java servlets or JavaServer Pages) where the page content is generated by the application server to incorporate customized data retrieved via multiple queries from the back-end database. This main HTML file is called a *server transaction*. Typically, the server transaction is responsible for most latency and consumed resources [6] (at the server side) during client transaction processing.

A client browser retrieves a web page (client transaction) by issuing a series of HTTP requests for all the objects: first it retrieves the main HTML file (server transaction) and after parsing it, the browser retrieves all the embedded, static images. Thus, at the server side, a web page retrieval corresponds to processing multiple smaller objects that can be retrieved either in sequence or via multiple concurrent connections. It is common that a web server and application server reside on the same hardware, and shared resources are used by the application and web servers to generate main HTML files as well as to retrieve page embedded objects.

Since the HTTP protocol does not provide any means to delimit the beginning or the end of a web page, it is very difficult to accurately measure the aggregate resources consumed due to web page processing at the server side. There is no practical way to effectively measure the service times for *all* page objects, although accurate CPU consumption estimates are required for building an effective application provisioning model. To address this problem, we define a *client transaction* as a combination of *all* the processing activities at the server side to deliver an entire web page requested by a client, i.e., generate the main HTML file as well as retrieve embedded objects and perform related database queries.

We use *client transactions* for constructing a "resource consumption" model of the application. The server transactions reflect the main functionality of the application. We use *server transactions* for analysis of the application performance changes (if any) during the application life-cycle.

3 Two Motivating Examples

Frequent software updates and shortened application development time dramatically increase the risk of introducing poorly performing or misconfigured applications to production environment. Consequently, the effective models for on-line, automated detection of whether application performance deviates of its normal behavior pattern become a high priority item on the service provider's "wish list".

Example 1: Resource Consumption Model Change. In earlier papers [20, 21], a regression-based approach is introduced for resource provisioning of multi-tier applications. The main idea is to use a statistical linear regression for approximating the CPU demands of different transaction types (where a transaction is defined as a client transaction). However, the accuracy of the modeling results critically depends on the quality of monitoring data used in the regression analysis: if collected data contain periods of performance anomalies or periods when an updated application exhibits very different performance characteristics, then this can significantly impact the derived transaction cost and can lead to an inaccurate provisioning model.

Figure 1 shows the CPU utilization (red line) of the HP Open View Service Desk (OVSD) over a duration of 1-month (each point reflects an 1-hour monitoring period). Most of the time, CPU utilization is under 10%. Note that for each weekend, there are some spikes of CPU utilization (marked with circles in Fig. 1) which are related to administrator system management tasks and which are orthogonal to transaction processing activities of the application. Once provided with this information, we use only weekdays monitoring data for deriving CPU demands of different transactions of the OVSD service. As a result, the derived CPU cost accurately predicts CPU requirements of the application and can be considered as a normal resource consumption model of the application. Figure 1 shows predicted CPU utilization which is computed using the CPU cost of observed transactions. The predicted CPU utilization accurately models the observed CPU utilization with an exception of weekends' system management periods. However, if we were not aware of "performance anomalies" over weekends, and would use all the days (i.e., including weekends) of the 1-month data set – the accuracy of regression would be much worse (the error will increase twice) and this would significantly impact the modeling results.

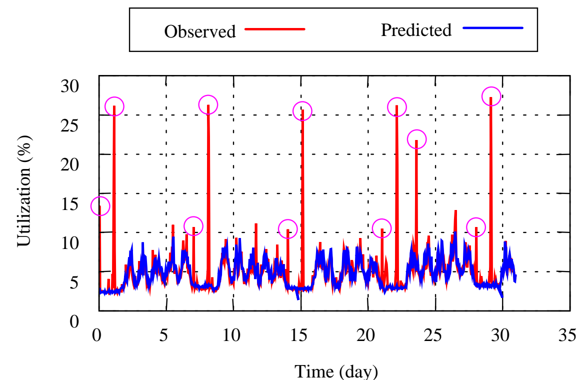

Figure 1. CPU utilization of OVSD service.

Example 2: Updated Application Performance Change. Another typical situation that requires a special handling is the analysis of the application performance when it was updated or patched. Fig. 2 shows the latency of two application transactions, $Tr1$ and $Tr2$, over time (here, a transaction is defined as a server transaction). Typically, tools like HP (Mercury) Diagnostics [13] are used in IT environments for observing latencies of the critical transactions and raising alarms when these latencies exceed the predefined thresholds. While it is useful to have insight into the current transaction latencies that implicitly reflect the application and system health, this approach provides limited information on the causes of the observed latencies and can not be used directly to detect the performance changes of an updated or modified application. The latencies of both transactions vary over time and get visibly higher in the second half of the figure. This does not look immediately suspicious because the latency increase can be a simple reflection of a higher load in the system.

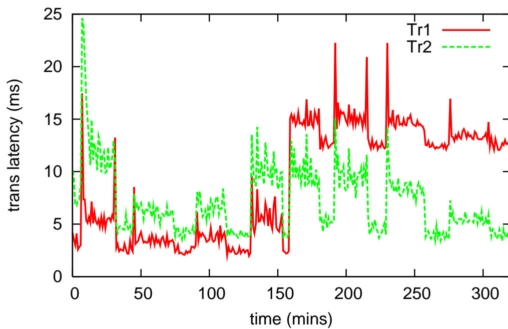

Figure 2. The transaction latency measured by HP (Mercury) Diagnostics tool.

The real story behind this figure is that after timestamp 160, we began executing an updated version of the application code where the processing time of transaction $Tr1$ is increased by 10 ms. However, by looking at the measured transaction latency over time we can not detect this: the reported latency metric does not provide enough information to detect this change.

Problem Definition. While using off-line data analysis we can detect and filter out the time periods that correspond to abnormal application performance or identify periods where application performance experiences significant change, the goal is to design an on-line approach that automatically detects the performance anomalies and application changes. Such a method enables a set of useful performance services:

- early warnings on deviations in expected application performance,
- raise alarms on abnormal resource usage,
- create a consistent dataset for modeling the application resource requirements by filtering out performance anomalies and pointing out the periods of changed application performance.

The next two sections present our solution that is based on integration of two complementary techniques: i) a regression-based transaction model that correlates processed transactions and consumed CPU time to create a resource consumption model of the application; and ii) an application performance signature that provides a compact model of run-time behavior of the application.

4 Regression-Based Approach for Detecting Model Changes and Performance Anomalies

We use statistical learning techniques to model the CPU demand of the application transactions (client transactions) on a given hardware configuration, to find the statistically significant transaction types, to discover the time segments where the resource consumption of a given application can be approximated by the same regression model, to discover time segments with performance anomalies, and to differentiate among application performance changes and workload-related changes as transactions are accumulated over time.

Prerequisite to applying regression is that a service provider collects the application server access log that reflects all processed client transactions (i.e., client web page accesses), and the CPU utilization of the application server(s) in the evaluated system.

4.1 Regression-Based Transaction Model

To capture the site behavior across time we observe a number of different *client transactions* over a monitoring window t of fixed length L. We use the terms "monitoring window t" or "time epoch t" interchangeably in the paper. The transaction mix and system utilization are recorded at the end of each monitoring window. Assuming that there are totally n transaction types processed by the server, we use the following notation:

- T_m denotes the time segment for monitored site behavior and $|T_m|$ denotes the cardinality of the time segment T_m, i.e., the number of time epochs in T_m;
- $N_{i,t}$ is the number of transactions of the i-th type in the monitoring window t, where $1 \leq i \leq n$;
- $U_{CPU,t}$ is the average CPU utilization of application server during this monitoring window $t \in T_m$;
- D_i is the average CPU demand of transactions of the i-th type at application server, where $1 \leq i \leq n$;
- D_0 is the average CPU overhead related to activities that "keep the system up". There are operating system processes or background jobs that consume CPU time even when there are no transactions in the system.

From the utilization law, one can easily obtain Eq. (1) for each monitoring window t:

$$D_0 + \sum_{i=1}^{n} N_{i,t} \cdot D_i = U_{CPU,t} \cdot L. \quad (1)$$

Let $C_{i,m}$ denote the approximated CPU cost of D_i for $0 \leq i \leq n$ in the time segment T_m. Then, an approximated utilization $U'_{CPU,t}$ can be calculated as

$$U'_{CPU,t} = C_{0,m} + \frac{\sum_{i=1}^{n} N_{i,t} \cdot C_{i,m}}{L}. \quad (2)$$

To solve for $C_{i,m}$, one can choose a regression method from a variety of known methods in the literature. A typical objective for a regression method is to minimize either the absolute error or the squared error. In all experiments, we use the Non-negative Least Squares Regression (Non-negative LSQ) provided by MATLAB to obtain $C_{i,m}$. This non-negative LSQ regression minimizes the error

$$\epsilon_m = \sqrt{\sum_{t \in T_m} (U'_{CPU,t} - U_{CPU,t})^2} \quad ,$$

such that $C_{i,m} \geq 0$.
When solving a large set of equations with collected monitoring data over a large period of time, a direct (naive) linear regression approach would attempt to set non-zero values for as many transactions as it can to minimize the error when the model is applied to the training set. However, this may lead to poor prediction accuracy when the model is later applied to other data sets, as the model may have become too finely tuned to the training set alone. In statistical terms, the model may "overfit" the data if it sets values to some coefficients to minimize the random noise in the training data rather than to correlate with the actual CPU utilization. In order to create a model which utilizes only the statistically significant transactions, we use stepwise linear regression [10] to determine which set of transactions are the best predictors for the observed CPU utilization. To determine the set of significant transactions, the stepwise regression algorithm initializes with an "empty" model which includes none of the transactions. At each following iteration, a new transaction is considered for inclusion in the model. The best transaction is chosen by adding the transaction which results in the lowest mean squared error when it is included. Before the new transaction is included in the model, it must pass an F-test which determines if including the extra transaction results in a statistically significant improvement in the model's accuracy. If the F-test fails, then the algorithm terminates since including any further transactions cannot provide a significant benefit. The coefficients for the selected transactions are calculated using the linear regression technique described above. The coefficient for the transactions not included in the model is set to zero.

Typically, for an application with n transactions, one needs at least $n+1$ samples to do regression using all n transactions. However, since we do transaction selection using a stepwise linear regression and an F-test, we can do regression by including only a subset of n transactions in the regression model. This allows us to apply regression without having to wait all $n+1$ samples.

4.2 Algorithm Outline

Using statistical regression, we can build a model that approximates the overall resource cost (CPU demand) of application transactions on a given hardware configuration. However, an accuracy of the modeling results critically depends on the quality of monitoring data used in the regression analysis: if the collected data contain periods of performance anomalies or periods when an updated application exhibits very different performance characteristics, then this can significantly impact the derived transaction cost and can lead to an inaccurate approximation model. The challenge is to design an on-line method that alarms service providers of model changes related to performance anomalies and application updates. Our method has the following three phases:

- *Finding the optimal segmentation.* This stage of the algorithm identifies the time points when the transaction cost model exhibits a change. For example, as shown in Figure 3, the CPU costs of the transactions $(Tr_1, Tr_2, ..., Tr_n)$ during the time interval (T_0, T_1) are defined by a model $(C_0, C_1, C_2, ..., C_n)$. After that, for a time interval (T_1, T_2) there was no a single regression model that provides the transaction costs within a specified error bound. This time period is signaled as having *anomalous* behavior. As for time interval (T_2, T_3), the transaction cost function is defined by a new model $(C'_0, C'_1, C'_2, ..., C'_n)$.

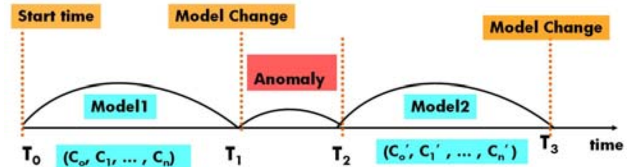

Figure 3. Finding optimal segmentation and detecting anomalies.

- *Filtering out the anomalous segments.* Our goal is to continuously maintain the model that reflects a normal application resource consumption behavior. At this stage, we filter out anomalous measurements identified in the collected data set, e.g., the time period (T_1, T_2) that corresponds to anomalous time fragment a shown in Figure 3.
- *Model reconciliation.* After anomalies have been filtered out, one would like to unify the time segments with no application change/update/modification by using a single regression model: we attempt to "reconcile" two different segments (models) by using a new common model as shown in Figure 4.

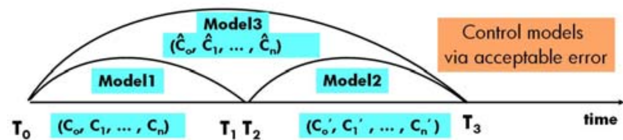

Figure 4. Model reconciliation.

We try to find a new solution (new model) for combined transaction data in (T_0, T_1) and (T_2, T_3) with a given (predefined) error. If two models can be reconciled then an observed model change is indicative of the workload change and not of an application change. We use the reconciled model to represent application behavior across different workload mixes.

If the model reconciliation does not work then it means these models indeed describe different consumption models of application over time, and it is indicative of an actual application performance change.

4.3 On-Line Algorithm Description

This section describes the three phases of the on-line model change and anomaly detection algorithm in more detail.

1) **Finding the optimal segmentation**
This stage of the algorithm identifies the time points where the transaction cost model exhibits a change. In other words, we aim to divide a given time interval T into time segments T_m ($T = \bigcup T_m$) such that within each time segment T_m the application resource consumption model and the transaction costs are similar. We use a cost-based statistical learning algorithm to divide the time into segments with a similar regression model. The algorithm is composed of two steps:

- construction of weights for each time segment T_m;
- dynamic programming to find the optimum segmentation (that covers a given period T) with respect to the weights.

The algorithm constructs an edge with a weight, w_m, for each possible time segment $T_m \subseteq T$. This weight represents the cost of forming the segment T_m. Intuitively, we would like the weight w_m to be small if the resource cost of transactions in

T_m can be accurately approximated with the same regression model, and to be large if the regression model has a poor fit for approximating the resource cost of all the transactions in T_m.

The weight function, w_m is selected as a Lagrangian sum of two cost functions: $w_{1,m}$ and $w_{2,m}$, where

- the function $w_{1,m}$ is the total regression error over T_m:

$$w_{1,m} = \sqrt{\sum_{t \in T_m} (U'_{\mathrm{CPU},t} - U_{\mathrm{CPU},t})^2} \quad ,$$

- the function $w_{2,m}$ is a length penalty function. A length penalty function penalizes shorter time intervals over longer time intervals to avoid dynamic programming to break the time into segments of very short length (since the regression error can be significantly smaller for a shorter time segments). It is a function that decreases as the length of the interval T_m increases. We set it to a function of the entropy of segment length as

$$w_{2,m} = -(|T_m|) \cdot \log(|T_m|/|T|)$$

Our goal is to divide a given time interval T into time segments T_m ($T = \bigcup T_m$) that minimize the Lagrangian sum of $w_{1,m}$ and $w_{2,m}$ over the considered segments, i.e., the segmentation that minimizes:

$$W_1(T) + \lambda W_2(T) \quad (3)$$

where the parameter λ is the Lagrangian constant that is used to control the *average* regression error ϵ_{allow} (averaged over T) allowed in the model, and

$$W_1(T) = \sum_m w_{1,m} \quad \text{and} \quad W_2(T) = \sum_m w_{2,m}.$$

Let us consider an example to explain the intuition for how the equation (3) works. Let us first consider the time interval T with no application updates or changes. Let time interval T be divided into two consecutive time segments T_1 and T_2.

First of all, $W_1(T_1) + W_1(T_2) \leq W(T)$, hence there are two possibilities:

- One possibility is that a regression model constructed over T is also a good fit over time segments T_1 and T_2, and the combined regression error of this model over time segments T_1 and T_2 is approximately equal to the total regression error over the original time interval T.
- The other possibility is that there could be different regression models that are constructed over shorter time segments T_1 and T_2 with the sum of regression errors smaller than a regression error obtained when a single regression model is constructed over T.

For the second possibility, the question is whether the difference is due to a noise or small outliers in T, or do segments T_1 and T_2 indeed represent different application behaviors, i.e., "before" and "after" the application modification and update.

This is where the W_2 function in equation (3) comes into play. The term $\log(|T_m|/|T|)$ is a convex function of $|T_m|$. Therefore, each time a segment is split into multiple segments, W_2 increases. This way, the original segment T results in the smallest W_2 compared to any subset of its segments, and λ can be viewed as a parameter that controls the amount of regression error allowed in a segment. By increasing the value of λ, we allow a larger W_1, regression error, in the segment. This help in reconciling T_1 and T_2 into a single segment representation T. In such a way, by increasing the value of λ one can avoid the incorrect segmentations due to noise or small outliers in the data T. When an *average* regression error over a single segment T is within the allowable error ϵ_{allow} (ϵ_{allow} is set by a service provider), the overall function (3) results in the smallest value for the single time segment T compared to the values computed to any of its sub-segments, e.g., T_1 and T_2. Therefore, our approach groups all time segments defined by the same CPU transaction cost (or the same regression model) into a single segment. By decreasing the value of λ, one can prefer the regression models with a smaller total regression error on the data, while possibly increasing the number of segments over the data.

There is a trade-off between the allowable regression error (it is a given parameter for our algorithm) and the algorithm outcome. If the allowable regression error is set too low then the algorithm may result in a high number of segments over data, with many segments being neither anomalies or application changes (these are the false alarms, typically caused by significant workload changes). From the other side, by setting the allowable regression error too high, one can miss a number of performance anomalies and application changes that happened in these data and masked by the high allowable error. [1]

2) **Filtering out the anomalous segments**

An anomalous time segment is one where observed CPU utilization cannot be explained by an application workload, i.e., measured CPU utilization can not be accounted for by the transaction CPU cost function. This may happen if an unknown background process(es) is using the CPU resource either at a constant rate (e.g., using 40% of the CPU at every time epoch during some time interval) or randomly (e.g., the CPU is consumed by the background process at different rates at every epoch). It is important to be able to detect and filter out the segments with anomalous behavior as otherwise the anomalous time epochs will corrupt the regression estimations of the time segments with normal behavior. Furthermore, detecting anomalous time segments provides an insight into the service problems and a possibility to correct the problems before they cause major service failure.

We consider a time segment T_m as anomalous if one of the following conditions take place:

- *The constant coefficient, $C_{0,m}$, is large.*
  Typically, $C_{0,m}$ is used in the regression model to represent the average CPU overhead related to "idle system" activities. There are operating system processes or system background jobs that consume CPU time even when there is no transaction in the system. The estimate for the "idle system" CPU overhead over a time epoch is set by the service provider. When $C_{0,m}$ exceeds this threshold a time segment T_m is considered as anomalous.
- *The segment length of T_m is short*, indicating that a model does not get fit to ensure the allowed error threshold.
  Intuitively, the same regression model should persist over the whole time segment between the application updates/modifications unless something else, anomalous, happens to the application consumption model and it manifests itself via the model changes.

[1] Appendix provides a formal description of the algorithm.

3) **Model reconciliation**
After anomalies have been filtered out, one would like to unify the time segments with no application change/update/modification by using a single regression model. This way, it is possible to differentiate between the segments with application changes from the segments which are the parts of the same application behavior and were segmented out by the anomalies in between. In such cases, the consecutive segments can be reconciled into a single segment after the anomalies in the data are removed. If there is an application change, on the other hand, the segments will not be reconciled, since the regression model that fits to the individual segments will not fit to the overall single segment without exceeding the allowable error (unless the application performance change is so small that it still fits within the allowable regression error).

4.4 Algorithm Complexity

The complexity of the algorithm is $O(M^2)$, where M is the number of time samples collected so far. This is problematic since the complexity is quadratic in a term that increases as more time samples are collected. In our case study, we have not experienced a problem as we used only 30 hours of data with 1-minute intervals, i.e., $M = 1800$. However, in measuring real applications over long periods of time, complexity is likely to become challenging. To avoid this, one solution might retain only the last X samples, where X should be a few orders larger than the number of transaction types n and/or cover a few weeks/months of historic data. This way, one would have a sufficiently large X to get accurate regression results, yet the complexity will not be too large.

5 Detecting Transaction Performance Change

Nowdays there is a new generation of monitoring tools, both commercial and research prototypes, that provide useful insights into transaction activity tracking and latency breakdown across different components in multi-tier systems. However, typically such monitoring tools just report the measured transaction latency and provide an additional information on application server versus database server latency breakdown. Using this level of information it is often impossible to decide whether an increased transaction latency is a result of a higher load in the system or whether it can be an outcome of the recent application modification and is directly related to the increased processing time for this transaction type.

In this section, we describe an approach based on an *application performance signature* that provides a compact model of run-time behavior of the application. Comparing new application signature against the old application signature allows detecting transaction performance changes.

5.1 Server Transaction Monitoring

Many enterprise applications are implemented using the J2EE standard – a Java platform which is used for web application development and designed to meet the computing needs of large enterprises. For transaction monitoring we use the HP (Mercury) Diagnostics [13] tool which offers a monitoring solution for J2EE applications. The Diagnostics tool consists of two components: the Diagnostics Probe and the Diagnostics Server as shown in Fig. 5.

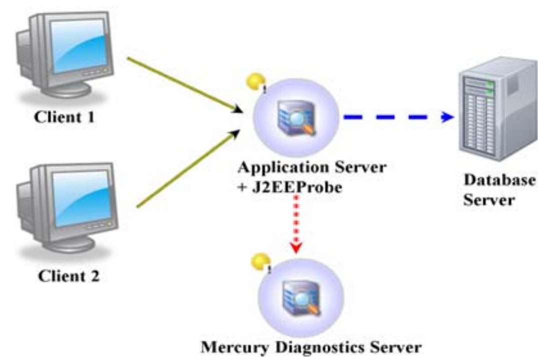

Figure 5. Multi-tier application configuration with the Diagnostics tool.

The Diagnostics tool collects performance and diagnostic data from applications without the need for application source code modification or recompilation. It uses bytecode instrumentation and industry standards for collecting system and JMX metrics. Instrumentation refers to bytecode that the Probe inserts into the class files of the application as they are loaded by the class loader of the virtual machine. Instrumentation enables a Probe to measure execution time, count invocations, retrieve arguments, catch exceptions and correlate method calls and threads.

The J2EE Probe shown in Fig. 5 is responsible for capturing events from the application, aggregating the performance metrics, and sending these captured performance metrics to the Diagnostics Server. We have implemented a Java-based processing utility for extracting performance data from the Diagnostics server in real-time and creating a so-called application log[2] that provides a complete information on all transactions processed during the monitoring window, such as their overall latencies, outbound calls, and the latencies of the outbound calls. In a monitoring window, Diagnostics collects the following information for each transaction type:

- a transaction count;
- an average overall transaction latency for observed transactions.[3] This overall latency includes transaction processing time at the application server as well as all related query processing at the database server, i.e., latency is measured from the moment of the request arrival at the application server to the time when a prepared reply is sent back by the application server, see Fig. 6;
- a count of outbound (database) calls of different types;
- an average latency of observed outbound calls (of different types). The average latency of an outbound call is measured from the moment the database request is issued by the application server to the time when a prepared reply is returned back to the application server, i.e., the average latency of the outbound call includes database processing and communication latency.

The transaction latency consists of the waiting and service times across the different tiers (e.g., Front and Database servers) that a transaction flows through. Let R_i^{front} and R_i^{DB} be the average latency for the i-th transaction type at the front

[2]We use this application log for building the regression-based model described in Section 4.

[3]Note that here a latency is measured for the server transaction (see the difference between client and server transactions described in Section 2).