Proceedings of the *8th IFIP/IEEE International Symposium on Integrated Network Management (IM 2003)*, Colorado Springs, CO, USA, March 2003

GENERIC ON-LINE DISCOVERY OF QUANTITATIVE MODELS FOR SERVICE LEVEL MANAGEMENT

Yixin Diao[1], Frank Eskesen[1], Steven Froehlich[1], Joseph L. Hellerstein[1] Alexander Keller[1], Lisa F. Spainhower[2], Maheswaran Surendra[1]

[1] *IBM T.J. Watson Research Center, P.O. Box 704, Yorktown Heights, NY 10598, USA*

{diao|eskesen|stevefro|hellers|alexk|suren}@us.ibm.com

[2] *IBM Server Group, 2455 South Rd., Poughkeepsie, NY 12601, USA*

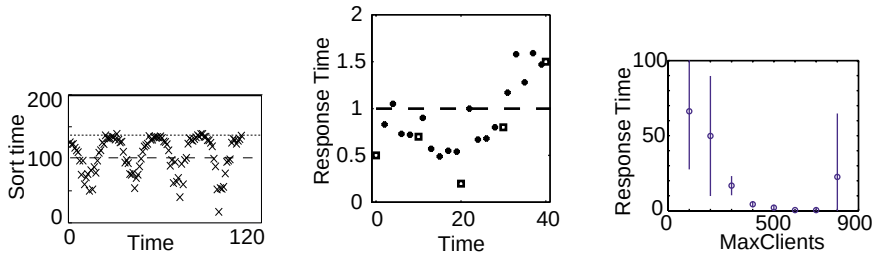
lisa@us.ibm.com

Abstract: Quantitative models are needed for a variety of management tasks, including (a) identification of critical variables to use for health monitoring, (b) anticipating service level violations by using predictive models, and (c) on-going optimization of configurations. Unfortunately, constructing quantitative models requires specialized skills that are in short supply. Even worse, rapid changes in provider configurations and the evolution of business demands mean that quantitative models must be updated on an on-going basis. This paper describes an architecture and algorithms for on-line discovery of quantitative models without prior knowledge of the managed elements. The architecture makes use of an element schema that describes managed elements using the common information model (CIM). Algorithms are presented for selecting a subset of the element metrics to use as explanatory variables in a quantitative model and for constructing the quantitative model itself. We further describe a prototype system based on this architecture that incorporates these algorithms. We apply the prototype to on-line estimation of response times for DB2 Universal Database under a TPC-W workload. Of the approximately 500 metrics available from the DB2 performance monitor, our system chooses 3 to construct a model that explains 72% of the variability of response time.

Keywords: Quantitative Model, Metric Discovery, Database System Instrumentation, Common Information Model, Service Level Management

1. Introduction

Central to service level management are tasks such as health monitoring to determine if the system is in a safe operating region, early detection of service level violations, and on-going optimization of configurations to ensure good performance. All of these tasks require quantitative insights, preferably quantitative models that predict service level metrics such as response time. Unfortunately, constructing such models requires specialized skills that are in short supply. Even worse, rapid changes in provider configurations and the evolution of business demands mean that quantitative models must be updated on an on-going basis. This paper proposes an approach to on-line discovery of quantitative models for service level management. The approach provides a way to construct quantitative models without prior knowledge of managed elements.



(a) Monitor health (b) Warn early (c) Optimize configuration

Figure 1. Some situations in which quantitative models is of value to service level management. (a) Identifying safe operating regions by using process control charts to track warning (horizontal line with long dashes) and critical (horizontal line with small dashes) limits on key variables, such as sort times in data bases. (b) Estimation of service level metrics such as response times (circles) provides early indications of service level violations compared to measured response times (squares). (c) Quantifying the impact of configuration parameters on response times provides a way to optimize configurations as illustrated by the parameter `MaxClients` for the Apache Web Server.

A variety of quantitative models are used in practice. For example, we relate DB2 performance metrics to response times using a model of the form $y = b_1 x_1 + b_2 x_2 + \cdots + b_n x_n$. Here, y is response time, the x_i are DB2 resource metrics (e.g., sort time, total buffer pool read time), and the b_i are constants estimated from the data using least-squares regression. We refer to y as the **response variable** and the x_i as the **explanatory variables**. Other examples of quantitative models include queueing network models (e.g., [12]), neural network models (e.g., [9]), and nearest neighbors approaches (e.g., [1]).

Figure 1 describes several situations in which quantitative models aid service level management. Considered first is monitoring system health by tracking key variables that characterize performance. For example, Figure 1(a) plots sort time, which turns out to be critical to the performance of eCommerce workloads such as those characterized by TPC-W [18]. A second situation is motivated by wanting an early warning of service level violations. This is illustrated in Figure 1(b) in which the squares are response times as measured by a response time probe. Such measurements are typically done at a low frequency because of cost and overheads. Augmenting these measurements with model-based estimates (the dots) provides a way to detect service level violations early (i.e., exceeding the dashed line). Last, optimizing configuration parameters requires having a quantitative understanding of how the parameters affect response times, which is greatly aided by having accurate quantitative models (especially for inter-related configuration parameters). Figure 1(c) illustrates this by showing that the Apache Web Server parameter `MaxClients` has a "U" shaped effect on response time, which permits using hill climbing to find a value of `MaxClients` that minimizes response time. Note that this method is applicable to any service level metric that is similarly affected by one or more configuration parameters.

While quantitative models can provide considerable value, their construction is difficult. Typically, a skilled analyst is required who understands the measurement data, configuration, and workloads. Changes in any of these or the relationships between them may mean that the model has to be reconstructed. This motivates a desire to automate model construction. Further, the approach should be *generic* in that it discovers the explanatory variables to use.

There are a number of areas of related work. Many researchers have investigated the detection of service degradations. Central to this is modeling normal behavior as in [13] which uses ad hoc models to estimate weekly patterns, [10] which employs more formal time series methods, and [19] which describes techniques for detecting changes in networks that are leading indicators of service interruptions. Further, statistical process control (SPC) charts are widely used for quality control in manufacturing [15] to detect shifts in a process as determined by an appropriate metric(s). These techniques have been applied to computing systems to track critical metrics (e.g., [14]). However, none of these approaches employ on-line model construction. More closely related to our work is [11], which uses knowledge of the functional relationship between inputs and outputs to detect changes in system operation. However, this work does not address how to identify a small set of explanatory variables. Several companies market products that aid in constructing performance policies (e.g., http://www.bmc.com). For the most part, the techniques employed are based on the distribution of individual variables, not relationships to response variables. One exception is correlation analysis (e.g., http://www.fortel.com), which uses cross correlation to identify the most important explanatory variables. However, this approach does not model the response variable. Thus, many redundant variables may be included. Further, all of the existing work assumes that the set of potential explanatory variables is known a priori rather than discovered on-line.

The remainder of this paper is organized as follows. Section 2 describes the architecture and information model used in our system for on-line metric discovery. Section 3 discusses the algorithms employed. Section 4 provides details of the prototype we built and illustrates its operation. Our conclusions are contained in Section 5.

2. System Overview

This section describes the architecture and information models used to support on-line discovery of quantitative models. Throughout, we assume that widely used supporting services are present, such as reliable message communication, persistence, registration, and location services.

2.1 Architecture

Figure 2 displays our architecture. The large rectangles identify key roles: the **Manager** and **Managed Element**. Although we treat all components in the architecture as objects, some can more naturally be thought of as data and others as procedures. The former are presented by ovals and the latter by small rectangles. Cascaded components (e.g., quantitative model) indicate that there may be several instances of them. The arrows indicate the flow of control or data, depending on whether the arrow connects two procedures or a procedure and data.

We begin by discussing the Managed Element. This is an encapsulation of a resource. A resource corresponds to functional entities such as a database, operating system, or web server. Our architecture augments the resource with an **Element Schema** that describes the resource (e.g., a database has tables and tablespaces), especially associated metrics (e.g., rows read, sort times) and configuration parameters (e.g., buffer pool sizes). In particular, we make use of the Common Information Model (CIM) [4] as a way to express the Element Schema. The **Agent Interface** (e.g., a

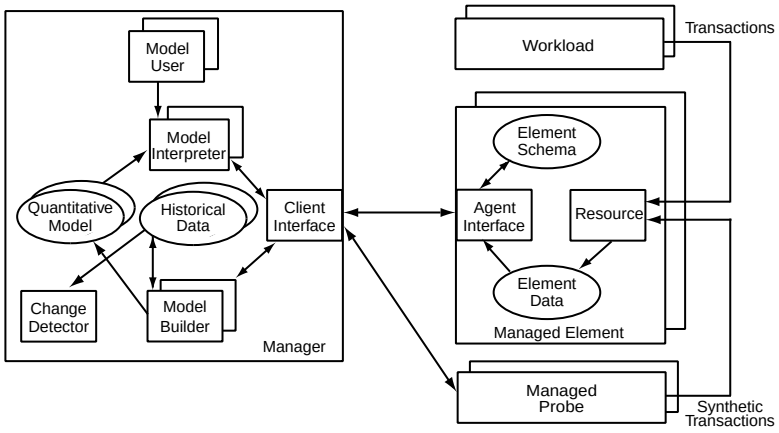


Figure 2. Architecture to support on-line discovery of quantitative models.

CIM Object Manager) is responsible for maintaining current values of element data as well as responding to requests to query the Element Schema. The use of an Element Schema is key to providing generic on-line discovery of quantitative models. Note that for our discussion, *schema* refers to the management information model describing the capabilities of a managed element, whereas the term *model* denotes the quantitative model used by the manager. Certain Managed Elements are used in special ways. A **Managed Probe** is a Managed Element that provides response time information by sending synthetic transactions to the resource and recording the transaction's start and completion times. Here, the Element Schema describes how to operate the probe (e.g., what synthetic transactions can be sent, the resource to which these transactions are sent) and the response times reported (e.g., by transaction type and resource).

Next we consider the Manager. The Manager communicates with Managed Elements through the **Client Interface**. The Manager has one or more **Model Users**, which are management applications that make use of quantitative models. Two examples of management applications are: (1) identifying key resource metrics for health monitoring and (2) predicting client response times for early detection of violations of service level agreements. For each type of model, there is a **Model Builder**. For example, we implemented a Model Builder that does real time construction of regression models [8]. The Model Builder typically requires historical data (e.g., to estimate the b_i in a regression model). These data are accumulated by using the Client Interface to find relevant data for the element (by querying the Element Schema) and then subscribing to updates of element data that are then placed in the Manager's historical data repository. Note that the same mechanism is used to acquire data from the managed elements and the managed probes. It may also be used to control workload generators if the Model Builder is conducting offline experiments to obtain a more diverse set of workload and system data. There are separate historical data for each model under construction. The **Model Interpreter** makes use of previously constructed quantitative models to estimate metrics of interest, such as estimating response times based on

resource internal metrics. The **Change Detector** component uses statistical techniques (e.g., [2]) or rule based policies to determine if model predictions deviate too much from actual values and therefore the model must be revised (possibly by re-invoking the Model Builder).

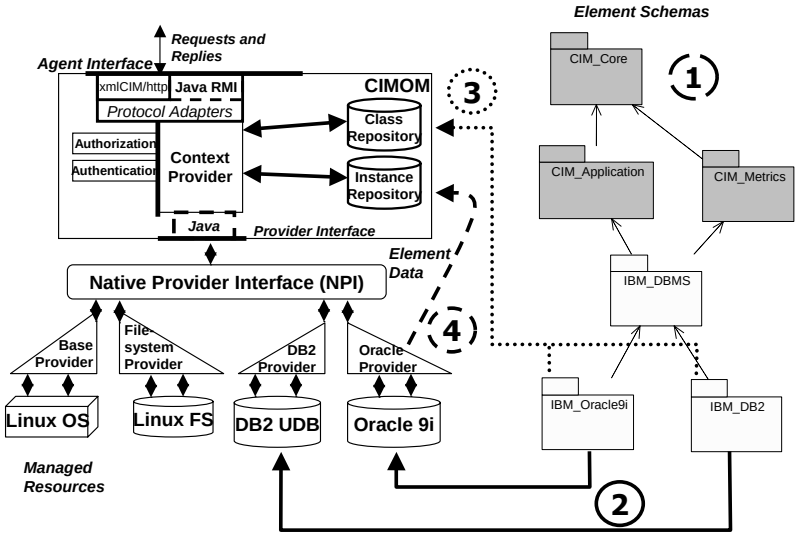


Figure 3. CIM Instrumentation of a Managed Element.

2.2 CIM-Based Managed Element

As mentioned earlier in Section 2.1, information about the Managed Element is surfaced by means of an agent based on the Web based Enterprise Management (WBEM) framework, whose core component is the Common Information Model (CIM) [4], a set of generic object oriented management models from which more specific resource models can be derived. The steps of designing and implementing a CIM based agent for retrieving metric data from the Managed Elements (in our case the database systems IBM DB2 UDB and Oracle9i) are depicted in Figure 3 and described below:

1 We identified the manageability information (e.g., descriptive and capability information, configuration parameters, statistical data such as counters and gauges etc.) available from our two target systems IBM DB2 UDB and Oracle9i. Then, we isolated the information common to both systems in an IBM_DBMS schema. The remaining data was placed into product-specific element schemas, which are named IBM_DB2 and IBM_Oracle9i, respectively (right side of figure).

2 CIM providers were implemented for the databases in accordance with the element schemas (solid lines at the bottom of the figure).

3 The standard CIM schema and the element schemas are loaded into the class repository of the CIM Object Manager (CIMOM). Then, the providers are registered with the CIMOM (as indicated by the dotted lines).

4 At runtime, clients request data from the Managed Element through the Agent Interface. These data are retrieved by the CIMOM, either from its instance repository or directly from the CIM providers (dashed line).

Designing CIM Extension Schemas for Database Systems. In accordance with CIM principles, every class of an extension schema is prefixed with a string identifying the creator of the management model (vs. the manufacturer of the resource); consequently, the classes we have introduced carry the prefix "IBM_". The hierarchy between the various CIM schemas and our extension schemas can be easily described by means of the UML package concept, which also captures dependency relationships (arrows between the various packages on the right side of Figure 3). It can be seen that we have used the CIM Core, Application and Metrics schemas as a basis to create our database models, i.e., we have derived the classes of our database models from classes defined in these schemas.

Designing the appropriate classes was fairly straightforward, given the fact that the basic architecture of a database system is common across multiple products: The overall `IBM_DBMS` (a subclass of `CIM_ApplicationSystem`) consists of a variety of subsystems, modeled either as subclasses of `CIM_SoftwareElement` or `CIM_SoftwareFeature`. In particular, we need classes to represent tables, tablespaces, buffer pools (aka caches), applications, and agents. The CIM schemas are designed according to the principle that all descriptive and capability-related information of a managed element is modeled as properties of the class representing the resource itself, while its statistical data is put in an associated class, subclassed from `CIM_StatisticalData`. This reflects the fact that the number of counters and gauges available for a managed element is in general fairly large, compared with the aforementioned descriptive resource information. Placing all statistical information into the resource class itself would lead to classes with many dozens of properties, which would result in an overly large amount of data that needs to be transferred if a class instance is to be retrieved. During our design, we were able to validate this assumption (some database parts have more than 30 different counters associated with them) and thus followed the recommended approach of encapsulating statistical information in separate classes. Overall, this leads to roughly a dozen leaf classes (plus a few association classes) that need to be implemented as CIM providers. In total, the classes of the IBM DB2 schema contain about 80 counters and gauges; for the sample database system we used for our experiments, this leads to about 500 metric instances that can be retrieved through the Agent Interface.

It should be noted that the DMTF Database Working Group [7] has recently published the first draft of a CIM Database Schema, which is likely to be included in the upcoming version 2.7 of the CIM Schemas. This schema contains a set of classes that describe a database in a very general way, which needs more refinement to be suitable for our purposes. More specifically, it defines three base classes `CIM_DatabaseSystem`, `CIM_CommonDatabase` and `CIM_DatabaseService` and three additional classes to capture statistical information related to the database as a whole and thus does not take the architectural details of a database system into account. Since our IBM_Database schema isolates the common elements of the two database systems with the largest market share, it is a possible candidate for submission to the DMTF Database WG for standardization.

Architecture of the CIM based Agent. The left part of Figure 3 depicts the architecture of our CIM agent. The part of the agent responsible for handling incoming requests and dispatching them to our providers is the publicly available SNIA (Storage Networking Industry Association) CIM Object Manager, which is implemented in Java. During our implementation, we encountered two important interoperability issues:

1 Currently, CIM does not specify the provider-facing part of a CIMOM; it is therefore not guaranteed that providers written for a specific CIMOM implementation are able to run with another CIMOM implementation.

2 Since the database systems expose their manageability information through interfaces in the C programming language, our providers are written in C, too. This brings up the need to interface between Java and C code.

The Native Provider Interface (NPI), implemented by the IBM Böblingen Laboratory as part of the SBLIM (**S**tandards **b**ased **L**inux **I**nstrumentation for **M**anageability) project [17], provides a convenient solution for both problems because it decouples the CIMOM from the various providers and the programming languages in which they are written. We use the NPI as glue code to interface between the Java based CIMOM and our database providers written in C. The detailed description of our prototype in Section 4 provides further details on our implementation experiences.

An object-oriented framework such as CIM permits the retrieval of all the statistical data in a single operation irrespective of the resource type (overall database system, tablespaces, tables, buffer pools, etc.). In particular, the CIM Operations over HTTP [5] protocol provides a means to enumerate all the instances of a given class. If the "deep" flag is set by a CIM client, this also applies to the instances of all the subclasses. Such a mechanism for retrieving only a selected subset of the available data is clearly superior to, e.g., a recursive "snmp-walk", which retrieves all of the data within a MIB (sub)tree. In our case, we can take advantage of this mechanism by retrieving all the instances of `CIM_StatisticalData` and its subclasses by means of a single operation. The CIMOM provides this level of abstraction by dispatching the requests to the appropriate providers, gathering the returned information and sending it back together, so that it is transparent to the client if the data surfaced by the CIMOM comes from one or many different CIM providers.

3. The Manager: Constructing Quantitative Models

This section describes the algorithms used by the Model Builder and Model Interpreter in Figure 2 for on-line construction and exploitation of quantitative models. To aid in this discussion, we use a running example in which DB2 UDB is the resource, **Element Data** are obtained from the DB2 performance monitor, and the response variable is response time (as measured by an external probe).

The **Quantitative Model** and algorithms employed by the Model Builder and Model Interpreter are specific to the modeling techniques employed. Thus far, we have found linear least-squares regression (e.g., [8]) to be very effective. The general form of a linear regression model is

$$y = b_0 + b_1 x_1 + b_2 x_2 + \cdots + b_n x_n .$$

The model relates the explanatory variables x_i to the response variable y through model parameters b_i. Linear models tend to be more robust than more elaborate non-

linear models such as neural network models. Also, linear least squares regression is widely applicable in solving real world problems. Moreover, building the linear models is less computational intensive so that it is more suitable for on-line model operation.

We begin by describing the content of the Quantitative Model object in Figure 2. For linear regression, it is sufficient to know the set of explanatory variables x_i and their associated constants b_i. For the running example, the challenge is determining a small number of metrics to use as explanatory variables out of the almost 500 metrics that can be obtained from the DB2 performance monitor.

It is the responsibility of the Model Builder to construct the Quantitative Model. Its inputs are `responseVariable`, the response variable (e.g., response time), and `candidateMetrics`, the set of candidate metrics. Its output is a Quantitative Model. Model Builder operates in two steps. The first is metric identification, which constructs `explanatoryVariables`, the set of explanatory variables x_i. The second step estimates the b_i.

3.1 Identifying Important Metrics

Metric identification determines which of the available metrics should be used as explanatory variables. Intuitively, it seems that using more explanatory variables results in a better model. However, this is often not the case. Indeed, extraneous variables may impair model performance when applied to new data.

Several approaches exist for selecting explanatory variables for quantitative models. Exhaustive search (e.g., false nearest neighbors [16]) examines all possible combinations of explanatory variables, but this is usually tractable only if the number of metrics is small. Ordered search explores the input possibilities according to certain importance measures, either in the incremented order or in the decremented order. One ordered search method is called stepwise regression. However, it may not necessarily produce the best model if there are redundant explanatory variables and may fail when applied to new data sets [8]. Moreover, it is computational intensive and may not be suitable for on-line discovery.

Metric identification takes as input `responseVariable` and `candidateMetrics` and produces `explanatoryVariables`, the set of explanatory variables. Metric identification begins once **Historical Data** have been collected for the candidate metrics. The details are described in Figure 4.

Note that the above algorithm does not consider all $2^m - 1$ possible models (where m is the number of metrics provided by the Managed Element). Rather, it incrementally selects the best metric based on what the current model does not explain. This simplifies the computation and makes it suitable for on-line discovery.

Also note that with more explanatory variables, the model tends to overfit the modeling data. To solve this problem, we use cross validation, a technique that employs testing data to assess model accuracy [20].

1 Initialization.
   (a) Set `candidateMetrics` to: list of metrics obtained from the Managed Element.
   (b) Set `residualVariable` to the response variable for the model.
   (c) Set `explanatoryVariables` to null.
2 Find the metric that best explains `residualVariable`.
   (a) Compute the cross correlation of each metric in `candidateMetrics` with `residualVariable`.
   (b) Set `bestMetric` to the metric with largest absolute value of the cross correlation.
   (c) Append `bestMetric` to `explanatoryVariables`.
3 Update variables.
   (a) Build a regression model of `responseVariable` on `explanatoryVariables` and set `residualVariable` to the residual of this model (the difference between the actual and estimated values of the `responseVariable`).
   (b) Remove `bestMetric` from `candidateMetrics`.
4 Check for termination.
   (a) Use cross validation to see if the testing error is increasing. If so
      i Remove `bestMetric` from `explanatoryVariables`.
      ii Return.
   (b) If `candidateMetrics` is not empty, then goto (2). Otherwise, return.

Figure 4. Algorithm for metric identification.

3.2 Estimating Parameters with a Quantitative Model

A wide range of standard techniques exist for estimating the regression model parameters b_i. In our case, the model is built initially using batch least squares. By "least squares", we mean that the unknown model parameters b_i are estimated by minimizing the sum of the squared deviations between the measured response data and the values estimated by the model. By "batch", we mean that all data are collected and *then* the b_i are estimated. The batch approach is well suited for off-line analysis, but can cause substantial computational overhead for on-line parameter estimation. As a result, we use recursive least squares [8], a technique that allows parameter estimates to be updated as new data are obtained. Not only does this reduce the computational overhead, it also provides a way to adapt to changes in workloads and configuration (although doing so requires another parameter, a forgetting factor, that determines the relative "weight" given to more recent data). To assess model accuracy, we employ the widely used R^2 metric.

$$R^2 = 1 - \frac{var(y - \hat{y})}{var(y)}$$

where y is the response variable, $\hat{y}$ is the estimated response variable, and $var(.)$ is the variance. The R^2 metric quantifies model accuracy by computing the variability explained by the model. R^2 ranges from 0 to 1. A value of 0 means the response data variability is not captured at all. A value of 1 may suggest a perfect fit.

4. Prototype Implementation

This section describes a proof-of-concept prototype constructed to demonstrate a generic approach to metric discovery. The prototype is depicted in Figure 5 and follows the architecture described in section 2.1. The Managed Element in the prototype is the IBM DB2 UDB Version 8.1 database management system running on a Linux

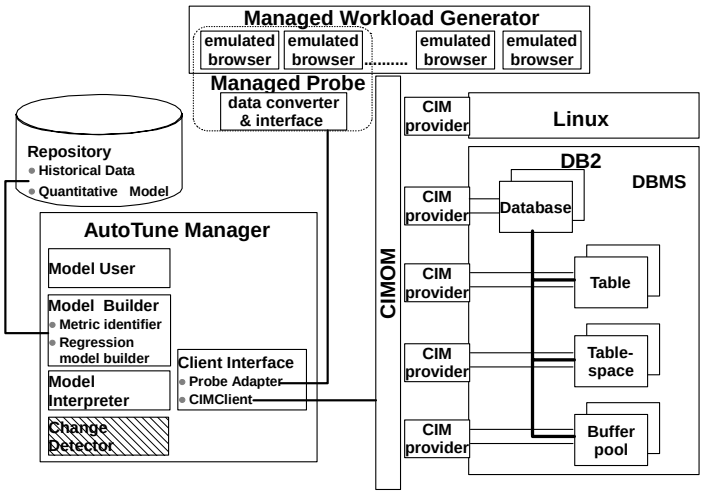


Figure 5. Architecture of the prototype

platform. As before, cascaded elements indicate multiple instances of an object (e.g. tablespaces, bufferpools).

4.1 CIM Providers for DB2 Performance Metrics

In order to dynamically obtain internal DB2 metrics, we developed CIM providers for DB2 Database, Bufferpool, Table and Tablespace information. Our providers use the `db2GetSnapshot` interface to obtain the data from DB2 and use the native provider interface to access the SNIA CIMOM.

This required some special code since the CIMOM environment requires that all functions operate in a thread-safe and thread-independent manner while the DB2 snapshot interface requires that all calls come from the same thread. The CIMOM can and often does create a new thread for each different request. Since each new CIMOM request would be driven from a separate thread and DB2 considers each thread a separate entity, without some sort of thread switching protocol within our provider, only meaningless (mostly zero) data could be extracted.

This led us to generate a separate thread for our DB2 application calls. All requests to DB2 are generated under control of this single separate thread. We atomically create this thread when we load the first provider and atomically delete it when we unload the last provider. When we receive a CIMOM request, we create a request element and atomically queue it to the DB2 thread. This request element contains a chain pointer, a synchronization mutex and a context descriptor. If, when queueing, the caller "atomically" recognizes that it is the first queuer, it also unlocks the DB2 accessor thread's mutex. After queueing the request, the caller waits for the DB2 thread to unlock the mutex contained in the queued request element. The DB2 thread unlocks the queue element mutex after the associated function completes.

The DB2 thread's logic is relatively simple. It has an input queue and a control mutex. The input queue is accessed atomically, using a locked compare and exchange instruction sequence. The first queuer, who changes the request queue from the empty state to the non-empty state, also unlocks the mutex after successfully changing the state. When the thread gains control, it atomically removes *all* elements from its input queue, thus changing the state back to empty. It then processes each removed element, one by one. After processing an element, it unlocks the mutex in the associated request element thus reactivating the calling thread. After processing all requests, the DB2 thread (again) waits for the mutex to be unlocked.

4.2 Workload Generation and Response Time Probes

The workload generator we use is TPC-W [18]. The characteristics of the workload are modulated by varying the number of emulated browsers (EB) and also the workload mix (buy vs browse). For the results shown here three types of workload mixes are used. The number of EBs is varied from 15 to 30 in a periodic fashion to approximately mimic time-of-day variations that are typical in an e-business environment.

For convenience, and also to take advantage of the measurement instrumentation available with the TPC-W kit, a subset of the EBs were also instrumented to provide client side response times (RT). Typically several transaction types are available from TPC-W, and for this work, the RT for the BestSellers transaction is used. On the same Linux system as the DB2 database, the workload was generated using emulated browsers each executing transactions according to the TPC-W benchmark specifications. A four hour workload cycle was created by continually increasing the number of emulated browsers from fifteen to thirty and then decreasing the number back down to fifteen. Additional variation was also introduced to the workload because of the probability associated with each emulated browser executing any one of the possible fourteen different transaction types. So analysis could be done later against the predicted response times, each active emulated browser logged the transaction type, the start time and completion time of that transaction.

4.3 Implementation of the Manager

The third major part of the prototype is the Manager. The Manager is built using the Agent Building and Learning Environment (ABLE) [3]. ABLE is a Java-based toolkit for developing and deploying hybrid intelligent agent applications. It provides a comprehensive library of intelligent reasoning and learning components packaged as JavaBeans (known as AbleBeans) and a lightweight Java agent framework to construct intelligent agents (known as AbleAgents). Built on top of ABLE is a general AutoTune Agent framework that facilitates the construction of on-line modeling/control agents. This general and extensible ABLE/AutoTune based implementation allows us to easily build the model, that is, the modules of Model User, Model Builder, and Model Interpreter. It also provides an interface to the Managed Element (for example, DB2) through an "adaptor" component that corresponds to the Client Interface in our architecture. Two adaptors are built: One communicates with Managed Element for DB2 and the other with the Managed Element for the response time probe. As an alternative to the latter, an adapter receiving SLA violation notifications from an SLA

monitoring system (described in [6]) can be used to determine how a user experiences the quality of service.

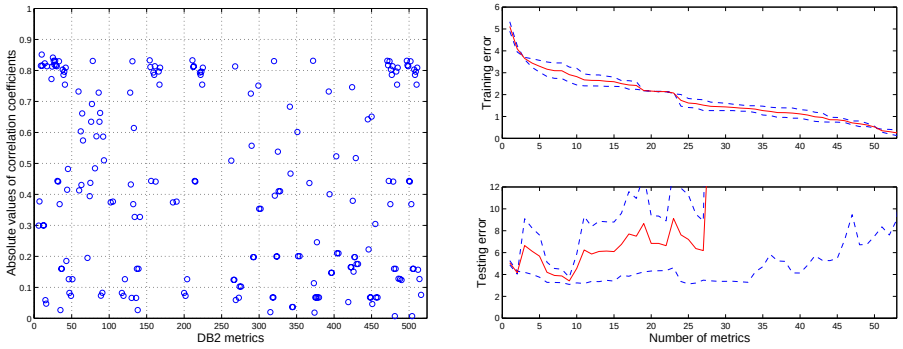
4.4 Interactions between Prototype Components

The operation of the prototype is as follows: First, the Model User instructs the Client Interface to obtain an enumeration of the available metrics from the CIMOM. In this implementation about 500 database metrics are made available. These metrics describe the operational status of the database across the multiple tables, tablespaces, and bufferpools. Although they provide a detailed view, most of them are either not directly related to predicting response time (e.g., `LockTimeouts` is a constant value which cannot be used to explain response time variations), or are essentially redundant. The metric identification procedure we described previously is able to discern the database metrics which are most effective as explanatory variables for response time.

Similarly, data from the response time probe are collected. The data are sent to the Historical Data repository. In our prototype, both database metrics and response time are averaged over 30 minute intervals.

Next, the Model Builder is invoked to do metric identification. To illustrate the effectiveness of the algorithm in Figure 4, Figure 6(a) displays the absolute values of the correlation coefficients between the DB2 metrics (about 500) and the probed client response times. The high values indicate there exists strong correlation between the DB2 metrics and the client response time, which argues for building a linear model for response time prediction.

Figure 6(b) displays how the root mean square of the residual changes with the number of explanatory variables used. The top plot is a result of metric identification on all data in Historical Data. We see that having more variables almost always improves the model. The bottom plot uses cross validation, which drops observations in order to have separate testing data. The plot indicates that only the first three metrics should be used as explanatory variables. The resulting model is



(a) Correlation coefficients (b) Determining key metrics

Figure 6. (a) Correlation coefficients between DB2 metrics and client response time. (b) Determine key metrics through cross validation.

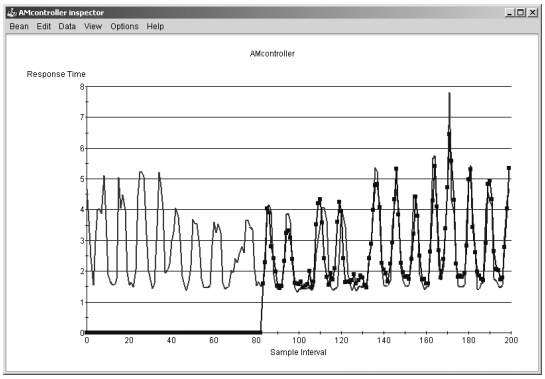


Figure 7. Screen shot showing extended time history of measured (light trace) and predicted (heavy trace) response time.

$$\begin{aligned} RT &= 1.44\texttt{ApplsExecutingInDBCurrently} \\ &+ 8.94 \times 10^{-5}\texttt{TotalBufferpoolReadTime} \\ &+ 9.69 \times 10^{-7}\texttt{TotalBufferpoolWriteTime} \end{aligned}$$

which has $R^2 = 0.72$ (i.e., explains 72% of the variability in the data). Since the workload variation is mainly caused by varying the number of emulated browsers, the ApplsExecutingInDBCurrently metric is identified as most important. The other two are also relevant because reading/writing to bufferpools is often where most of the delay in the database occurs when processing queries. The relative importance of these metrics, which are identified here without any in-depth knowledge of DB2, is consistent with the expectations of experienced database administrators.

Once the model is built, the Model User instructs the Model Interpreter to use the model to predict RT based on incoming DB metrics. The effectiveness of the model is apparent from Figure 7, where the light trace is RT as reported by the probe, and the heavy trace is the predicted RT. Note that in the left part of the plot, the heavy trace is flat since this is the period when the data are being collected for model building.

5. Conclusions and Outlook

Quantitative models have considerable value in performance management. Unfortunately, constructing quantitative models requires specialized skills that are in short supply. Even worse, rapid changes in provider configurations and the evolution of business demands mean that quantitative models must be continuously updated.

This paper proposes an approach to on-line discovery of quantitative models that operates without prior knowledge of managed elements. In particular, the Common Information Model (CIM) is used to discover metrics exposed by managed elements. These metrics are input to an algorithm that selects a subset to use as explanatory variables and then builds a quantitative model. Our approach employs an architecture in which the Managed Element includes components that describe the resource (Element Schema), contain data collected from the resource (Element Data), and an interface to

the manager (Agent Interface). The manager has a matching Client Interface used by the Model Builder, which constructs quantitative models, and the Model Interpreter, which runs the models. The approach is demonstrated for estimating response times for DB2 UDB with a TPC-W workload. We show that of the approximately 500 metrics (counters and gauges) available from the DB2 performance monitor, our system chooses 3 to construct a model that provides very good estimates of response times.

While our initial results are encouraging, much work remains. Currently, the response variable (e.g., response time, throughput) must be known when the Model Builder is invoked. We are extending our architecture to include extracting response variables from a service level agreement specification. Another direction is to adapt the model on-line, such as when there are changes in workloads and/or configuration (which may require change-point detection). Last, we want to scale our techniques to address multi-tiered eCommerce systems.

References

[1] J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger. Adaptive algorithms for managing a distributed data processing workload. *IBM Systems Journal*, 36(2), 1997.

[2] M. Basseville and I. Nikiforov. *Detection of Abrupt Changes: Theory and Applications*. Prentice Hall, 1993.

[3] J. P. Bigus, D. A. Schlosnagle, J. R. Pilgrim, W. N. Mills III, and Y. Diao. ABLE: A toolkit for building multiagent autonomic systems. *IBM Systems Journal*, 41(3), 2002.

[4] Common Information Model (CIM) Version 2.2. Specification, Distributed Management Task Force, June 1999. http://www.dmtf.org/standards/cim_spec_v22/.

[5] Specification for CIM Operations over HTTP, Version 1.0. Specification, Distributed Management Task Force, August 1999. http://www.dmtf.org/download/spec/xmls/CIM_HTTP_Mapping10.php.

[6] M. Debusmann and A. Keller. SLA-driven Management of Distributed Systems using the Common Information Model. In G.S. Goldszmidt and J. Schönwälder, editors, *Proceedings of the 8th IFIP/IEEE International Symposium on Integrated Network Management*. Kluwer Academic Publishers, March 2003.

[7] *DMTF Database Working Group*. http://www.dmtf.org/about/working/database.php.

[8] Frank E. Harrell. *Regression Modeling Strategies: With Applications to Linear Models, Logistic Regression, and Survival Analysis (Springer Series in Statistics)*. Springer Verlag, 2001.

[9] Simon Haykin. *Neural Networks: A Comprehensive Foundation*. Macmilan College Publishing Company, 1994.

[10] P. Hoogenboom and J. Lepreau. Computer system performance problem detection using time series models. In *Proceedings of the Summer USENIX Conference*, 1993.

[11] R. Isermann and B. Freyermuth. Process fault diagnosis based on process model knowledge. In *Proceedings of 1989 ASME International Computers In Engineering Conference and Exposition*, July 1989.

[12] Leonard Kleinrock. *Queueing Systems Volume I*. Wiley-Interscience, 2nd edition, 1975.

[13] Roy A. Maxion. Anomaly detection for diagnosis. In *Proceedings of the 20th International Annual Symposium on Fault Tolerance (FTCS)*, June 1990.

[14] J. McConnell, D. Helsper, L. Lewis, and S. Joyce. Predictive analysis: How many problems can we avoid? In *Networld+Interop, Las Vegas*, 2002.

[15] D.C. Montgomery. *Introduction to Statistical Quality Control*. Wiley, 3rd edition, 1997.

[16] Carl Rhodes and Manfred Morari. Determining the model order of nonlinear input/output systems. *AIChE Journal*, pages 151–163, 1998.

[17] *Standards Based Linux Instrumentation for Manageability Project*. http://oss.software.ibm.com/developerworks/projects/sblim/.

[18] Wayne D Smith. TPC-W: Benchmarking an ecommerce solution. In *http://www.tpc.org/tpcw*.

[19] Marina Thottan and Chuanyi Ji. Adapative thresholding for proactive network problem detection. In *IEEE Third International Workshop on Systems Management*, April 1998.

[20] S. Wold. Cross-validatory estimation of the number of components in factor and principal components model. *Technometrics*, 20(4):397–405, 1978.