Abstract

This paper describes a novel, flexible framework, eModel, designed to address the runtime requirements of autonomic computing: on-line workload measurement, analysis, and prediction. The eModel architecture has been developed using platform independent technology (XML and Java) to allow for maximum portability while also allowing for ease-of-integration with existing measurement and system management tools. The eModel toolkit consists of a GUI based model builder tool, a data base deployment tool, a runtime tool, and an analysis tool. In addition to the toolkit, the eModel design provides a runtime architecture which can be deployed directly without using any interaction with the GUI. The architecture is flexible enough to allow for incorporation with models of various complexity, including modeling techniques that require a hierarchical approach to attain reasonable accuracy based upon on-line, measured data. We present examples that illustrate eModel as a capacity planning tool as well as an augmentation to autonomic system management in an effort to highlight the technological gaps that the eModel framework is capable of bridging.

1. Introduction

In autonomic computing, where each system component monitors its own performance data, and accordingly takes necessary actions, (making it self-healing, self-configuring and self-optimizing) closer integration of various analytic tools for modeling its online performance with the underlying runtime systems is a requirement. However, traditional focus on tooling has been on developing sophisticated off-line or on-line tools capturing as much of the details of an environment as possible ([2], [1]). Integrating these types of tools as run-time component of a system demands that we pay as much attention to the integration framework as to the models themselves. Ease-of-use continues to remain an important issue, however, not necessarily only as an issue in running the tool by a human operator but also as an issue in setting up the tool for system development/deployment. The current paper introduces a flexible architecture for such a modeling framework, and demonstrates diverse usage scenarios for different objectives and/or environments.

An autonomic system component typically monitors and reconfigures itself to comply with service level agreements (SLAs see [9] for an example SLA specification) on its usage, established with the clients of this system. SLAs are established by clients with the service provider systems in order to receive a guarantee on various service level objectives, e.g., average response time for supported throughput level during certain time periods, availability of services, etc. The eModel architecture is used during all phases of a SLA life-cycle: creation, deployment and runtime monitoring & enforcement. For more details on the SLA life-cycle implementations see [5].

In all of the SLA life-cycle scenarios, the overall analysis/modeling tool needs to support the following features. First and foremost, it needs to provide ease-of-use in setting up the modeling framework, where a model may be a single plug-in or composed from a collection of plug-ins in tandem. From the point of view of on-line integration of this modeling framework with varied data sources (e.g., database tables, online calls to runtime systems, etc.), we need flexible ways of specifying data sources to be used and their access details. Similarly, the eModel framework needs to support flexible ways of delivering analysis data to other runtime system components (e.g., via database, direct calls, etc.). A detailed set up describing the usage of specific model(s) can be specified declaratively using an XML document, which can be created via a GUI. Then a runtime
program can parse the XML document and create the corresponding on-line modeling structure and appropriate hooks into measurement and management tools.

This paper is outlined as follows. Section 2 describes the system design requirements that were used to drive the eModel development. Section 3 gives an overview of the eModel architecture. Examples of the eModel implementation are given in section 4. Finally, we give a summary of future work in section 5.

2. Design Goals

The three main objectives in the eModel system design are ease-of-use, flexibility, and scalability. In the following paragraphs we describe these design goals in more detail.

Ease-of-use: We require that the framework should be easy to use for both the model provider and the model user. The eModel design should include a workload measurement and model code repository which can be easily implemented and expanded. Furthermore, although prediction engines that are employed may include sophisticated, state-of-the-art statistical mathematics, neural networks, etc., we want the internal rigors of the models to be transparent to the end-user. More specifically, any model parameters (i.e. approximation order, prediction horizon, confidence intervals) that must be set prior to runtime estimation should have default values set by the model code writer while a guideline for changing these parameters is available if the user chooses to experiment. Similarly, the methodology to integrate existing measurement and modeling tools and applications should be straightforward, with minimal impact to an application developer deploying eModel on a system.

Flexible: The eModel runtime architecture should also be flexible enough to adapt to improving modeling techniques. For instance, several researchers ([6], [7], [10], [8] to name a few) have shown that web traffic is of a hierarchical nature with a combination of stationary and non-stationary characteristics; the time-series of a typical web server workload trace consists of hourly, daily, and weekly trends. Therefore, the model techniques used to construct estimates of web traffic would consist of components to predict at each timescale and intelligent methods to combine the different forecasting levels. This type of modeling and analysis leads to design requirements that allow for a sequence of models to be invoked with room for interaction between the different models in the sequence.

Scalable: The eModel architecture should also be able to track and forecast multiple workloads of multiple types (of both service level and application request type) from possibly several remote locations in a heterogeneous environment. Of course, since eModel is a real-time/on-line forecasting tool, the numerical methods required for workload modeling and estimation must strike a balance between accuracy, complexity and efficiency with the ultimate design point being able to run all algorithms on a thin-client architecture.

In order to incorporate all of these design goals, we will further require a rich yet concise language to describe the workload to be analyzed. For instance, we need to know: what quantities need to be measured and how often the online samples should be taken; what are the important features in the time-series (workload) we are modeling (i.e. periodicity); what is the input and output as well as necessary parameters for the models that are being used; and what type of prediction horizon and confidence interval should be used in data forecasting, and should we take any action based upon a prediction (i.e. alert or event generation).

3. Architecture

3.1 User Defined Components and eModel Interfaces

As previously stated, one of the major design goals of the eModel architecture is flexibility - we wanted the eModel runtime engine (see figure 1) to easily fit in with existing measurement and system management tools. Therefore, we did not hardwire in any specific measurement or management agents, servlets, or methods nor did we seek to directly modify system parameters once any event conditions were triggered. The following sections provide some insight into how a developer writes code to connect to the eModel architecture. Examples of all of the interfaces are discussed in section [5].

Figure 1. The eModel architecture.
3.1.1 Measurement Service

Inherent to the idea of analyzing and predicting workload behavior is a systematic manner of gathering measurements to build workload models. We do not define a specific methodology to gather such data. Instead we define a Java interface class which a user can implement to connect the native data collection methods to the eModel runtime engine. This interface class is called a MeasurementService and it has methods for initialization and data collection which are used by the eModel runtime engine.

3.1.2 Event Handling

Predicted workload metrics can be used to enhance SLA system management utilities, such as request scheduling and policy based provisioning. Examples of such an SLA based management system are given in [3] and [4]. In order to act upon these predicted values, threshold conditions must be evaluated and events triggered based upon the results of the evaluation. The eModel architecture does not define a specific event checking engine. Rather, as with the MeasurementService class, we have written an Event Java interface class which users can implement to adhere to their own event monitoring system. The user-defined event class is instantiated at startup time and used not only as the condition evaluator but also as the action invoker.

3.1.3 The Model Class

We want eModel to be used to track a variety of workloads, of which each type may be more amenable to a specific modeling paradigm (i.e. linear regression versus neural nets). Moreover, the actual modeling that is done within the eModel engine is implementation specific. In fact, we want to support any set of black boxes which take a given set of inputs (Measurement objects) and return a given set of outputs (again Measurement objects). Our decision to use an object representation of data was deliberate; in this fashion, anything that can be written as a Java object (integers, double precision numbers, strings, etc.) can be used as input or generated as output for models. Therefore, we have defined a Model Java interface class. This class has methods for building a model given a set of measured data, i.e. calculating coefficients in an autoregression model, as well as deriving predictions from the model for a given time horizon.

3.1.4 Application Interface to eModel Data

Measurement and prediction data from the eModel runtime are available in cached eModel process memory as well as in a persisted database. For application writers who want to access the data, we provide two static classes. WorkloadAnalysis has methods to retrieve data from the eModel process, i.e. this is how to obtain the data when the eModel runtime engine is active. WorkloadRegistry provides a set of methods for users to access the persisted database without doing any calls (JDBC, SQL, etc.) to the database.

3.2 eModel Components

As has been eluded to the previous sections, the eModel code suite is Java-XML based. This allows us to develop eModel in a platform independent manner. Developers using native methods for data collection, workload modeling, or event handling must keep this in mind as they integrate with the eModel interfaces. In the following sections we discuss the various eModel components in more detail.

3.2.1 Model Building

The first step in tracking any workload is to build a workload descriptor file. In this file we must include measurement information, model information, and output information. The XML schema implemented to contain this information as well as an example SLA is given in [5]. The GUI used to create, edit and save descriptor files is called the ModelBuilder in the eModel architecture and is described in more detail in [5].

3.2.2 Deployment and Database Tools

Once an workload descriptor file has been generated, it must be stored in the database for future use. The eModel Deployment tool is the interface to the database for this purpose. Once a file has been selected for deployment, the deployment tool parses the XML file and creates an object. Object fields are then persisted in the database as table records. These table records are used by the runtime engine (section 3.2.3) to build (or “hydrate”) runtime workload objects. Direct access by the user or application developer to the database for any of the workload object fields or data is not necessary. We provide a static Java class, WorkloadRegistry, to read or write data to the database.

3.2.3 Runtime Engine

The eModel runtime engine has two main parts, a container (the Manager) and runtime objects (the Workloads). At its initialization, the Manager looks in the database for all Workload objects that need to be built, builds the object from the database entries, and then creates an extensible thread pool, binding a single Workload object to a thread.
The Manager maintains a mapping between Workload objects and names (the WorkloadId field from the descriptor file) and running threads. The Manager then initializes each of the Workload runtime objects, including the measurement method, each of the model methods, and each of the event methods for the corresponding object.

The Workload object is set to tracking mode either with the Runtime eModel GUI interface by clicking the Start button or by calling the Manager.start(name) method directly with the appropriate workload name. Once the workload runtime object is tracking it has four sequential functions which are run continuously: get a measurement; loop through the model methods and build a model; loop through the model methods to make a prediction; fire all event methods; and store data to the database.

To date, we have only assessed the robustness of the Manager’s runtime engine with a limited set of workloads. For these workloads, the linear chaining of models and read/write model parameter and DB utilities have been sufficient for accurate analysis and predictions. We expect that as more complex workloads are unveiled and studied, the runtime engine will have to address this complexity, perhaps by adding a feedback function, or a function to send data from one model to multiple other models. The complexity of the runtime model engine for robustly managing a variety of workload models is currently an area that we are researching.

3.2.4 Analysis Tools

The Analysis tool within the eModel framework is a real-time animation of measured and predicted values versus time as shown in figure 2. Several workloads and metrics can be plotted on separate frames at one time. Since the measured and predicted data are also available to the application developer with the WorkloadAnalysis and WorkloadRegistry classes, customized analysis tools can be built.

4. Examples

We now illustrate two uses of the eModel framework.

4.1 Capacity Planning

The example illustrates on-demand and predictive provisioning of a high volume web site. eModel framework provides all the essential components, i.e., a real time prediction engine, mechanisms to collect data from on-line load measurement system, and a event generation mechanism based on monitored/predicted data.

Based on the sufficient advanced warning (e.g., hours, exact duration can be programmed in eModel), customer can dynamically configure servers (i.e., add hardware) with the increasing demand for capacity. We assume that each of the web server nodes of the high volume web site maintains a log of web requests, and eModel MeasurementService class is implemented as a programmable load measurement system that reads the web logs, aggregates data from multiple nodes and calculates a request rate. The prediction engine invokes the getMeasurement() method to obtain the total number of requests received for all servers since the last measurement is collected and a load metric is calculated (i.e. Requests/hour). The frequency of this invocation is programmable as per the requirements of prediction engine. We have adopted the methodology of flexible metrics specification using XML as outlined in [9].

Next, we construct a model of our web site traffic to be used by the prediction engine. Let’s assume that the web traffic on our site has strong weekly and daily patterns (or periodicity). For instance, we know that peak traffic occurs on all days from noon to 4pm, and from 4am to 8am the traffic is at its lowest point. We will therefore need to build a weekly “template” of the traffic pattern. We do this by dividing the week into states where each state corresponds to a different hour that has been measured – e.g. 168 states per week. Further we assume that the time series that each state history represents is stationary. Using this assumption we can build a time series of data for each state and apply traditional statistical methods for predicting the load values of each state in the future. For this example, we have chosen to use an autoregression method with a lag of 4 (i.e. 4 weeks). Using this methodology, with every measurement that is taken the measurement is mapped to the appropriate state in the week and added to that state’s time series vector and AR coefficients are generated for that state using the time series data. Using the AR coefficients, we can generate a weekly template 1 week, 2 weeks, etc. ahead.

We know that the shape of the template may remain the same from week to week (i.e. Wednesday is a peak day, Saturday and Sunday are off-peak), but certain changes can effect the overall day-to-day volume. One can think of this as shifting the volume under the curve (or template) up or down. We therefore need short term modifications to the template for improved accuracy. In this example, we use a simple volume ratio scaling based upon the most recently taken measurements. The algorithmic details are as follows:

1. we calculate a ratio \( r = \frac{\text{template}}{\text{current}} \) for the last \( n \) hours (states) where the load (\( \lambda \)) estimate is taken from our weekly template estimating method;

2. we take an average of the \( n \) ratios;

3. we multiply the template estimates the next \( n \) hours
by this average ratio – this becomes the new forecast for these hours.

This two-step estimation process is a methodology; various statistical techniques could be used to generate both the state estimate (i.e. ARMA and exponentially weighted moving average (EWMA)) and the daily scaling (i.e. EWMA with filtering). Furthermore, we could sample the log file more frequently and generate the statistical structure of each state. However, for the example that we are concerned with, these techniques proved to be sufficient for results typical of the web server capacity example (see figure 2).

Figure 2. Results from the prediction methodology described in section 4.1 for two different B2c websites (“Customer19” on top; “Customer4” on the bottom). Here we plot requests/hour versus time in a 2 week time interval. The lightly colored line is estimated values; the darker line is actual measurements. Note the periodical nature of the time series (period is 1 week).

Finally, generation of events (say increase in weekly load) based on monitored and predicted data is also programmable in eModel. Lets assume that we’ve set an event in the descriptor file to send an e-mail to sysadmin@bighost.com whenever the traffic is predicted to be above $\zeta$ request per hour. Now, whenever the eModel runtime engine constructs a model using the methods described above and forecasts a new set of values, the checkEvent() method is called MailMe.java that checks the values of the prediction and sends the e-mail that the event has occurred. The system administrator has received an internal alert that a change in the system configuration may be required well ahead of the actual change so that appropriate action may be taken. We have adopted the syntax for condition specification as outlined in [9].

4.2 Provisioning & SLA Based System Management

This example differs from the previous one in that we are interested in shared resources (i.e. the same web servers being used to host multiple customer sites) and redistributing the workload or re-allocating resources autonomically. Again, we will need a measurement system and a prediction engine, but in this case we are sending the forecasted values to SLA aware system management via the Event tag in our descriptor. Due to space constraint, the details are omitted in this paper and can be found in [5].

5. Summary and Future Work

eModel is a flexible modeling framework that can be integrated with existing runtime systems for addressing various requirements of autonomic computing: on-line workload measurement, analysis and prediction. Modeling an activity requires not only specifying how and what are to be collected/used for online analysis but also a sequence of analysis operations (including prediction and detection of conditions) to be performed on this data and delivering the result(s) to the intended runtime components. In eModel an XML based specification describes all of the above aspects of modeling an activity. We have developed a (Java and XML based) portable toolkit consisting of a GUI based model builder that stores the deployed models in a database, and a runtime environment that performs online data collection, analysis and result delivery as per the specification of deployed models. We presented several examples that illustrate eModel as a capacity planning tool as well as an augmentation to autonomic system management in an effort to highlight the technological gaps that the eModel framework is capable of bridging.

The future work on eModel has two distinct, parallel paths: one is to continue to improve the eModel architecture including building an extensive code repository of models for all types of workloads, and second, is to apply this technology in various usage scenarios described in this paper.

6. Acknowledgements

We are grateful to Li Zhang, Cathy Xia, and Zhen Liu of the Systems Analysis & Optimization group at IBM Research for all of their helpful suggestions regarding web traffic analysis and prediction which have been used in the capacity planning example (section 4.1) We are also appreciative to Dan Dias of IBM Research and Noshir Wadia of the IBM High Volume Web Sites group for their feedback on the eModel system design and architecture.
References