ADAPTIVE MONITORING IN ENTERPRISE SOFTWARE SYSTEMS

Mohammad Ahmad Munawar and Paul A.S. Ward
Shoshin Distributed Systems Group
Department of Electrical and Computer Engineering
University of Waterloo, Waterloo, Ontario N2L 3G1, Canada
E-mail: {mamunawa,pasward}@shoshin.uwaterloo.ca

ABSTRACT

Today's enterprise software systems are mission critical. Ideally they must run continuously, correctly. To do otherwise costs money. However, system failures will still occur. Continuous system monitoring can reduce the time to diagnosis, but comes at the cost of reduced system efficiency. One alternative, partial system monitoring, is to monitor at a minimal level to determine system health, and adaptively increase the monitoring level if some problem is suspected. Given the complexity of modern enterprise servers, rather than develop an explicit model based on system knowledge, we employ simple statistical techniques to identify relationships in the monitored data. These relationships are used to characterize normal operation and, in the event of anomalies, identify areas that need more monitoring. In this paper we motivate the need for adaptive monitoring, we describe an approach to drive this adaptation, and present preliminary results of our adaptive-monitoring prototype.

1 INTRODUCTION

Many businesses today rely heavily on software systems to support operations and provide services to a large user-base. These systems are generally expected to be operational around the clock and provide services to a large user-base. Failures in this context can cost a great deal either in the form of lost opportunities or penalties for failing to meet service-level agreements. These systems comprise a set of system and application software, each of which fulfills specific needs. Each software can itself be composed of many parts and components. A software system to support an online store, for example, includes one or more end-user applications, operating systems, database servers, HTTP servers, application servers, workload routers, *etc*.

Enterprise information systems are required to meet stringent reliability requirements. Proper systems management is thus of utmost importance. However, management has become hard as these systems are increasingly large, have complex structure, and are frequently updated. The composite and componentized structure of these systems entails an overwhelming number of entities that need to be taken into consideration for management purposes. In addition, relationships among parts and components need to be understood because slight changes may lead to important disturbances [8].

Despite the availability of large amounts of information regarding many aspect of modern software systems, effectively managing their operation is hard. Each software in a system can be monitored *via* numerous performance, activity, resource utilization, and state-related metrics. Sources include log files, trace files, event notifications, existing instrumentation and related interfaces, dynamic instrumentation, *etc*. These systems can generate an overwhelming amount of information that can be costly to collect [6] and difficult to handle and analyze [9]. Traditionally, the problem of system management has been tackled by employing enough human experts who are effective in finding relevant information and reasoning under uncertainty. The state of the art in systems monitoring is that a small set of important information is always monitored; the rest is only collected when system administrators deem it necessary. Current tools help administrators sort through large amount of information, but do not eliminate their involvement. This state of affairs calls for novel approaches to systems management. An effort in this respect is Autonomic Computing [9]. Many aspects of Autonomic Computing such as self-configuring, self-healing, self-tuning, self-protecting are dependent on effective system monitoring.

In this paper we motivate the need for automated adaptive monitoring. The monitoring system should continuously assess current conditions by observing the most relevant data; it must promptly detect anomalies; it should help identify root-causes of problems. For an intelligent monitor to observe the most relevant data, it must adapt (*i.e.*, it should dynamically enable collection

and analysis of information that would be most useful in reducing uncertainty). It should adapt to prevailing conditions and collect just enough information to ensure correct and efficient operation of the system under observation. Collecting less information not only reduces the adverse effect of measurement on system performance, but also reduces the overhead associated with storing, transmitting, analyzing, and reporting information. Adaptive monitoring also limits the probe effect on the measurement [7]. For example, we are likely to get more precise values of some pertinent metrics if others are not collected. Thus, the system can run in conditions as similar as possible to the minimally instrumented version.

Today there is no self-adaptation of monitoring. Significant time is spent detecting problems. System administrators manually adjust when and what information needs to be collected. They may fail to correctly assess problematic situations due to limited knowledge and may lack the ability or availability to respond promptly. Increasing the number of administrators is expensive and does not necessarily solve the problem. Skilled and knowledgeable administrators are in short supply.

2 Background

To support scalable and distributed business applications, ease of development, and software re-use, standardized component-frameworks have been developed and widely adopted. These frameworks provide commonly-needed services such as directory, transactions, remote communication, security, *etc.* and implement features such as resource-pooling, multi-threading, synchronization, *etc.* Such services and features are bundled in server platforms and shared by the various applications.

One of these frameworks is Java 2 Enterprise Edition (J2EE [3]). J2EE specifies application-program interfaces (API) and interactions for basic services needed for distributed and enterprise computing. It also defines interfaces, roles, and deployment details of components in the framework. A J2EE application is a combination of many specialized components. Some components such as Servlets and Java Server Pages (JSP) handle the presentation logic. Others such as Enterprise Java Beans (EJB) deal with the business logic. The J2EE server, comprising component containers and various services, provides the common runtime environment for individual J2EE applications.

We can monitor most of the components of any application running on a J2EE server. Information on web components, such as servlets, may comprise the number of requests being served over time or at any given instance, number of errors encountered, response-time, *etc.* As far as enterprise beans are concerned, depending on the type of bean, different information can be gathered. In general, one could monitor how many instances of each bean have been created, number of active beans, number of free beans available in various pools, average response-time per bean, number of times the various methods of a bean are called, number of times a bean is persisted, time taken to persist a bean, *etc.* Interestingly, information as detailed as the time taken by a particular method of a bean can be measured.

The primary tool for learning and anomaly detection used in this work is linear regression. Regression analysis attempts to find a model that relates a dependent variable to one or more independent variables. In the case of simple linear regression, model parameters (*i.e.*, slope and intercept) can be estimated by fitting a line that minimizes the sum of square of the difference between actual and estimated values. The correlation coefficient can be used to check how well the regression line fits the data. This coefficient varies between -1 and 1. The closer it is to zero, the worse is the fit. More details on linear regression can be found in any introductory book on statistics.

3 Related Work

Significant recent research work has focused on prompt detection and diagnosis of problems in enterprise systems (*e.g.,* [10]). The proposed approaches often assume that detailed system-related data is available. However, in practice, such data is typically too expensive to collect.

Seltzer and Small [13] have looked at adaptive monitoring in the context of extensible operating systems. Likewise, it has been investigated in the context of dynamic code instrumentation for monitoring program execution flow and performance (*e.g.*, [5, 11]).

Irina *et al.* [12] describe an approach to inferring system state by adaptively selecting and executing the most informative probe. The authors use Bayesian networks to encode the relationship between tests' outcomes and states of nodes in the system. The test results are then used to update the model parameters, which can be used to query the belief on the current system's state.

Our approach has some similarity to that of Brown *et al.* [4]. Their goal, however, is to infer dependencies between components of a system by actively inducing perturbation in the system. They also use statistical correlation to learn the strength of the dependencies identified. In contrast, our work assumes that some level of perturbance always exists in modern software systems due to their dynamic nature. Moreover, we use regression estimates to predict values of metrics using other metrics.

4 Our Approach

The continuous observation of runtime metrics related to activity, state, load, or performance of a software system gives rise to time-series data. Analyzing this data to distinguish deviations from normal behaviour is non-trivial.

Modeling the metrics individually is difficult as they are frequently non-linear. Rather, we propose to model them in relation to other metrics, especially metrics that are closer to each other in sequences of dependencies. This makes modeling simpler by obviating the need to consider many sources of non-linearity.

Our approach consists of four phases: system modeling, minimum monitoring, adaptive monitoring, and fault diagnosis. In the system modeling phase we do three things. First, we determine all pairs of metrics originating from distinct subsystems that are linearly correlated. This may be performed by maximal system monitoring for a period of time, collecting sufficient quantity of data to allow for cross testing of all pairs. In practice, we use *a priori* knowledge to identify dependence among subsystems at a high level. Knowing high-level dependencies reduces the number of metric-combinations that need to be tested for correlation. Second, we estimate regression parameters for the correlated pairs. Initially this is performed together with the first step, though as the software executes for a long period of time we have found that parameters' estimates need to be updated. Third, a small number of low-collection-cost metrics needs to be identified and modeled individually. These metrics act as basic health indicators, enabling us to determine when are where to increase monitoring levels.

In the second phase, monitoring is reduced to a minimal level, collecting only those health-indicator metrics identified in phase one. Anomaly detectors are associated with each such health indicator. When an anomaly is found, the adaptive monitoring logic takes control, and the system enters phase three. Monitoring is increased such that metrics known from phase one to be correlated with the anomalous health-indicator metric are retrieved. Now the data collected maps to a set of metric-pairs. When monitoring metric-pairs, an anomaly occurs when the predicted value of the dependent metric noticeably deviates from what the linear regression model predicts. When such anomalies are found, metrics correlated with the out-of-bound metric are collected and checked for anomalies. Thus the process continues recursively.

Finally, the process of increasing the monitoring level, *i.e.*, retrieving more metrics, is stopped either when there is sufficient evidence to identify a faulty component or when no more evidence can be gathered even though no fault has been found. In the latter case, a false positive has occurred, suggesting that the monitoring system may need parameter adjustment. The stopping criteria can be implemented in different ways and is a research problem which we are trying to address.

In phase four, diagnosing the fault, we rank components according to the number of times their metrics are reported as being out-of-bound. Components that are higher-ranked are more likely to be the cause of the observed anomalies for two reasons. First, a faulty component is likely to exhibit multiple aberrant forms of behaviour, which may cause it to be reported multiple times. Second, because most components have correlated metric pairs with multiple other components, a faulty component will be reported many times. Information from different metrics is thus corroborated, reducing the likelihood of a wrong diagnosis.

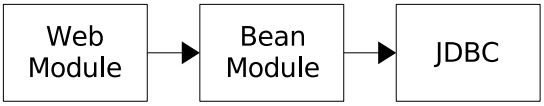

Figure 1: Subsystems dependency for identifying correlated metric-pairs

The environment of the target system changes often (*e.g.*, surge in workload). As such, regression parameters learned initially may no longer apply after some time. These parameters can periodically be refreshed by sampling values of the different metrics in a way that does not heavily impact performance. For example, we could iterate over the available metrics in a round-robin fashion such that in each period a subset of metrics are collected and the corresponding parameters adjusted. When one cycle completes all metrics would be refreshed.

5 Implemented Prototype

We have implemented a prototype based on our proposed approach. Our experimental setup consists of a DB2 UDB 8.1 database server, a WebSphere 5 Application Server (WAS), a custom-made workload generator, and the prototype monitoring engine, each executing on a separate machine. Running the monitoring engine on a separate machine reduces its impact on the performance of the target system. All machines are connected via a Gigabit LAN. We use the Trade 3 [1] benchmarking end-user application, which implements an online stock brokerage system comprising Servlets, JSPs, and EJBs.

The monitoring engine collects data available from the Performance Monitoring Interface (PMI) of WAS. This data reflects activity, state, errors, and performance of components and services in the server. The monitor polls for data it needs every 10 seconds. The WAS version used only provides coarse-grained control over what metrics are collected. Five levels of monitoring are available: *None, Low, Medium, High, and Maximum*. Each level corresponds to a set of metrics, including those at lower levels. A typical J2EE application has three important subsystems: a Web module containing Servlets and JSPs; a Bean module containing different types of EJBs; a JDBC driver dealing with connection to the back-end database server. Fig. 1 depicts the dependencies among these subsystems for Trade 3.

Using previously collected data and following the dependency depicted in Fig. 1, we determine which pairs of metrics are linearly correlated. Each such pair originates from two connected subsystems. Only metric pairs with a strong linear correlation are considered. We checked correlation between metrics and estimated linear regression parameters using the R [2] statistical package.

5.1 Operation of the Monitoring Engine

At startup, the monitoring engine is provided with the list of correlated metrics determined above. The monitoring engine sets the monitoring level to maximum for all metrics related to the three subsystems. It then begins to estimate linear-regression parameters for correlated metrics. We use an estimation period that is deemed sufficient to obtain stable parameters. This period defines a sliding window over which estimates are calculated. We wait for the system operation to reach a steady state. When this happens, the monitor sets the monitoring level to minimum. At this level, we monitor only response-time of, and errors from, web components (*i.e.,* Servlets and JSPs), which act as our health indicators. This maps to a monitoring level of *Medium* for the Web module and *None* for the other modules.

The health-indicator anomaly detectors are defined as follows. In case of response-time metrics, we maintain averages over a fixed-size sliding window. An alarm is raised when an observed value exceeds the running average by more than a certain factor. In the case of Servlet errors, when the observed value exceeds a certain threshold, an anomaly is declared.

When a health-indicator anomaly is detected, the monitor sets the monitoring level to *Medium* for all three subsystems. Using the data thus collected, it checks whether observations made at the present monitoring level are within reasonable bounds of those predicted by the models learned. If they are not, the monitor makes use of rules to try to identify the subsystem responsible for the problem. At each step, we make use of four measures: the number of Web-Bean pairs presently available (WB_A), the number of Web-Bean pairs where the observed value is out-of-bound (WB_O), the number of Bean-JDBC pairs presently available (BJ_A), and the number of Bean-JDBC pairs where the observed value is out-of-bound (BJ_O). We then compute two ratios: $WB_R = WB_O/WB_A$ and $BJ_R = BJ_O/BJ_A$. If WB_R is higher than a threshold A and BJ_R is lower than a threshold B, then we attribute the problem to the Web module. This rule is based on the reasoning that a problem in a web component will mostly affect the Web-Bean pairs. A problem in a web component can indirectly affect Bean-JDBC pairs. Threshold B is used to take these effects into consideration. If BJ_R is more than threshold B and WB_R is less than than threshold A, then the JDBC subsystem is considered as the source of the problem. If both WB_R and BJ_R are higher than a threshold B, then the Bean module is considered to be causing the problem. This is because problems in beans affect both Web-Bean and Bean-JDBC pairs. If none of these rules apply, the monitor increases the monitoring level for all three parts of the application. It then computes the ratios mentioned above and tries to apply the above rules. Once a subsystem is assigned responsibility for the anomaly, or the maximum level of monitoring is reached, problem determination ends. Otherwise, the monitor steps up the monitoring level.

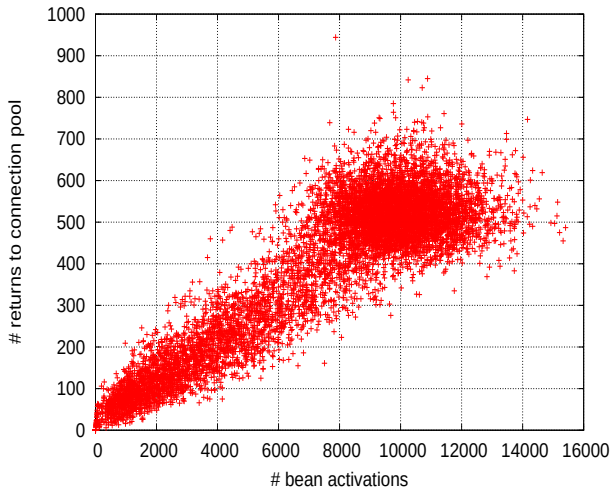

Figure 2: Example of a correlated metric-pair

The adaptive monitoring logic of the prototype differs from the proposed approach because of limitations in the granularity of monitoring available in the current testbed. Instead of finding metrics correlated with those that are out-of-bound, we simply increase monitoring so as to have more metric-pairs to analyze. We empirically derived thresholds A and B. More work is needed to define these objectively and in a more systematic way.

In order to find more detailed information than subsystem-level, we implemented the component-ranking approach described in Section 4. With this approach, we can easily shortlist components that are likely to be causing problems.

5.2 Experiments

As a first step of experimentation, we collected most available metrics from the web module, bean module, and JDBC component of the Trade3 application over a period of roughly 24 hours. We then ran a set of scripts to identify correlated pairs. Fig. 2 presents an example of two linearly correlated metrics.

In order to evaluate the effectiveness of our alarm triggers as well as the monitor's responses to alarms, we

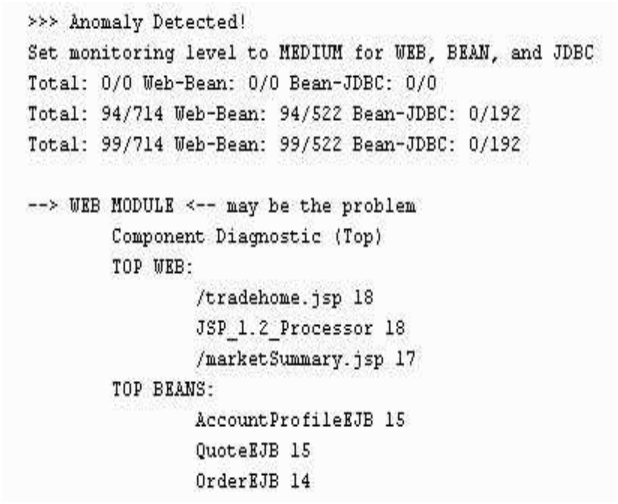

Figure 3: Example of output printed by the Monitoring Engine

injected a number of synthetic faults in the Trade3 application and created other anomalous conditions. Our experiments were mainly based on two types of fault in two locations. The types were delay and exceptions. Exceptions were controlled by specifying the probability of occurrence of an exception. These faults can be added to one of the Servlets, JSPs, or EJBs. Delays were controlled specifying the duration. Fault were injected either in the application (*i.e.*, in one of the Servlets, JSPs, or EJBs) or in the environment. Faults injected in the environment focused on either CPU consumption on one of the machines or simulated network errors.

In most experiments the anomalous component percolated to the top three of the list of potentially problematic components. Attributing the problem to a particular subsystem using rules described earlier, however, was trickier. While initially the engine correctly pinpoints the problematic subsystem, as errors propagate, it makes mistakes. Figure 3 depicts the output from the user interface of the Monitoring Engine after having introduced exceptions in a component of the `tradehome` webpage.

Based on these preliminary experiments, we have validated the basic approach, while identifying a number of shortcomings and research questions which we expect to address. First, our approach does not allow for monitoring of metrics that are not linearly correlated with other metrics. We have observed that such metrics do exist. Second, all metrics are not equal. Some metrics are more important than others insofar as they affect more metrics than others. As such, diagnosis based on the number of times a metric is reported as being anomalous does not always shortlist the faulty component. Third, when monitoring is reduced to the minimum level, it is possible for the environment to change. We have yet to implement parameter updating as proposed in Sec. 4.

6 Conclusions and Future Work

In this paper we argue that adaptive monitoring is critical for effective systems' management. We propose an approach for monitoring component-based enterprise systems using statistical correlation between metrics. We describe a prototype that partially implements our approach and report early results from our experience.

We are porting the prototype to use a testbed with a more recent version of the application server, which allows for fine-grained control of the collection of metrics. As far as the approach is concerned, we are looking at ways to accommodate metrics that do not correlate with others, keeping learned relationships up-to-date, incorporating metrics' weight in the diagnosis, and dealing with changes in system-state that result from the occurrence of anomalies. In the long run, we would like to move beyond simple linear regression and use other techniques that would capture more relationships between metrics. Moreover, this work only considers a single application. Application servers can support many application at a time. As such, we need to extend the proposed approach to cater for such cases.

References

[1] Trade3. http://www3.ibm.com/software/webservers/appserv/-benchmark3.html.

[2] The R Project for Statistical Computing. http://www.r-project.org/.

[3] Sun Microsystems Inc. J2EE 1.4 Platform Specification. Available at http://java.sun.com/j2ee/j2ee-1_4-fr-spec.pdf.

[4] Brown, A., Kar, G., and Keller, A. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. In *Proc. of IEEE Int'l Symposium on Integrated Network Management* (May 2001), pp. 377–390.

[5] Dmitriev, M. Profiling Java applications using code hotswapping and dynamic call graph revelation. In *Proc. of the 4th int'l Workshop on Software and Performance* (2004), pp. 139–150.

[6] Fox, A., and Patterson, D. Self-repairing computers. *Scientific American* (June 2003).

[7] Gait, J. A probe effect in concurrent programs. *Software — Practice and Experience 16*, 3 (Mar. 1986), 225–233.

[8] Gribble, S. D. Robustness in complex systems. In *Proc. of the 8th Workshop on Hot Topics in Operating Systems* (2001).

[9] Kephart, J., and Chess, D. The vision of Autonomic Computing. *IEEE Computer 36*, 1 (January 2003), 41–50.

[10] Kiciman, E., and Armando, F. Detecting application-level failures in component-based internet services. *IEEE Transactions on Neural Networks 16*, 5 (September 2005), 1027–1041.

[11] Mirgorodskiy, A. V., and Miller, B. P. Autonomous analysis of interactive systems with self-propelled instrumentation. In *Proc. of the 12th Multimedia Computing and Networking*.

[12] Rish, I., Brodie, M., Ma, S., Odintsova, N., Beygelzimer, A., Grabarnik, G., and Hernandez, K. Adaptive diagnosis in distributed systems. *IEEE Transactions on Neural Networks 16*, 5 (September 2005), 1088–1109.

[13] Seltzer, M., and Small, C. Self-monitoring and self-adapting operating systems. In *Proceedings of the 6th Workshop on Hot Topics in Operating Systems* (1997), p. 124.