Statistical Approaches to Predicting and Diagnosing Performance Problems in Component-based Distributed Systems: An Experimental Evaluation

Sand Correa
Department of Informatics
PUC-Rio, Brazil
Email: scorrea@inf.puc-rio.br

Renato Cerqueira
Department of Informatics
PUC-Rio, Brazil
Email:rcerq@inf.puc-rio.br

***Abstract*—One of the major problems in managing large-scale distributed systems is the prediction of the application performance. The complexity of the systems and the availability of monitored data have motivated the applicability of machine learning and other statistical techniques to induce performance models and forecast performance degradation problems. However, there is a stringent need for additional experimental and comparative studies, since there is no optimal method for all cases. In addition to a deeper comparison of different statistical techniques, studies lack on two important dimensions: resilience to transient failures of the statistical techniques, and diagnostic abilities. In this work, we address these issues, presenting three main contributions: first, we establish the capability of different statistical learning techniques for forecasting the resource needs of component-based distributed systems; second, we investigate an analysis engine that is more robust to false alarms, introducing a novel algorithm that augments the predictive power of statistical learning methods by combining them with a statistical test to identify trends in resources usage; third, we investigate the applicability of statistical tests for identifying the nature and cause of performance problems in component-based distributed systems.**

I. Introduction

Current distributed applications exhibit complex behaviors stemming from the interaction of numerous hardware and software elements, workload, traffic conditions, and quality guarantees. To handle this complexity, middleware standards, such as CORBA, Java RMI, and Web Services, have been used to provide reusable services across distributed applications. Such services include security, persistence, and distributed connectivity. The middleware approach clearly separates the business logic specific to each application from the common services required across multiple systems. The modularity and reusability provided by this solution successfully address complex system requirements. Nonetheless, managing the performance of the emerging systems becomes more difficult, as the complete system behavior is observed only during execution. In addition, middleware technologies have a significant runtime footprint, and the quantification of this overhead relies on the infrastructure and implementation provided by each technology.

In fact, it is recognized that middleware technologies make the problem of understanding the eventual performance of distributed applications even more difficult [1]. Despite this fact, providing performance guarantees is a crucial issue for many applications. Therefore, many approaches have been proposed to deal with this problem, most of which try to decrease the burden on system management by developing models for capturing the dynamics of middleware-based applications. For example, much work [2], [3], [1] has been done in analytical modeling of system performance, in which *queuing network* and *control theory* have been applied to model the behavior of applications built on middleware technologies. However, these models may require substantial effort from human experts and may be hard to derive, as middleware-based systems exhibit a complex relationship with the execution environment. On the other hand, the current monitoring and data collection tools allow measurements on the various components of a distributed system. The availability of monitored data has inspired some recent work [4], [5], [6] to apply statistical learning techniques (methods, algorithms) to build performance models automatically. These approaches do not assume human involvement and require little domain analysis.

Despite all efforts to apply statistical learning techniques in the context of performance problems, the use of these techniques for autonomic system management is still at an early stage and there is considerable room for improvement, since most efforts concentrate on the forecasting dimension, and evaluations are taken for individual techniques. However, as observed by Kohavi in [7], no algorithm can outperform all others in every case and, thus, more comparative studies are important. The relevance of such studies is also recognized by Zhang et al. in [8], in which the authors highlight the need for reviewing the applicability and suitability of different techniques for the autonomic management field (see Section IV). Beyond the forecasting dimension, few efforts have applied statistical learning techniques for diagnostic purposes. Nevertheless, the main drawback of this approach is that many machine learning techniques are not easily interpretable, and this fact may turn the diagnosis more difficult.

In this paper, we present an experimental evaluation

of the capability of statistical approaches to characterize performance problems in middleware-based systems. More specifically, the presented experiments are structured in order to answer three questions:

- **Q_1**: Which statistical learning methods are suitable to forecast whether the system will meet some objective within a given time span, based on a current set of observed metrics?
- **Q_2**: Considering that statistical learning methods are subject to false alarms, can we make the prediction process more robust against transient failures of the learning methods?
- **Q_3**: Whenever a period of poor performance is forecast, can we quantify the influence of each monitored metric on this undesirable event?

The answers to these questions will support the analysis of performance problems in terms of three dimensions that are relevant for any autonomic performance management tool: prediction of performance problems, robust prediction, and diagnosis. The first dimension enables rational allocation of computing resources and better task scheduling. The second dimension attempts to reduce the false alarm rate of prediction methods, ensuring more robustness to predictions under production settings. The third dimension enables the identification of the cause of a problem and allows specific adaptation decisions to be taken to repair the problem.

We implemented a vast number of experiments for which a reference scenario was defined: a MapReduce application built on top of a component-based middleware system. The experiments enabled this work to make the following contributions:

- A systematic comparison of learning methods applied to predict performance degradation in a middleware-based system. We examine the abilities of the major families of classifiers in providing an online mechanism to capture the performance behavior of middleware based-systems.
- The design and evaluation of an analysis engine that is more robust to false alarms. We present a performance degradation warning algorithm that augments the predictive power of machine learning methods, combining them with a statistical test for trend. To the best of our knowledge, this is the first warning algorithm designed to achieve robustness using such combination of approaches.
- An evaluation of statistical tests applied to identify the nature and cause of performance problems. Statistical tests are simple, can be computed very efficiently and also provide a way to promote diagnosis regardless of the statistical learning method in use by the forecasting activity. To the best of our knowledge, statistical tests have not been used for diagnosing performance problems in any previous work.

This paper is organized as follows: in Section II, some additional discussion about performance management in the context of this work are provided, and the reference scenario is introduced; in Section III, the experiments and their results are presented; Section IV discusses related work; and Section V concludes and describes future work.

II. The Reference Scenario

This work focuses on ensuring high performance of middleware-based applications, which is defined in terms of business objectives or service level objectives (SLO). An example of a typical service level objective is a threshold on the response time of a service, as this metric is directly linked to customer satisfaction. With respect to an SLO, a service can provide one of two states: compliance or violation. A service is said to be in compliance with an SLO if the performance goal is met; otherwise, it is said to be in violation. In this work, we assume that a service in violation of an SLO is indicative of poor performance.

In order to analyze the performance behavior of middleware-based systems, we defined a reference scenario, in which SCS [9], a CORBA-like component-based middleware, was used as a reference to middleware technology. In SCS, a service is represented by one or more application level components. Two elements of the runtime infrastructure control the overall deployment of the application components. The first element, known as *component container*, is responsible for controlling the lifecycle and execution of application components. The second element, known as *execution node*, represents the network node in which component containers and application components are deployed. As described in previous work [10], SCS is instrumented with monitoring points which measure system (execution node), middleware (container), and application (application component) metrics.

As a reference to component-based applications, we considered a MapReduce [11] framework implemented using SCS components. MapReduce is a master-worker parallel programming style, used in clusters of computers for processing huge data sets on certain kinds of distributable problems. Throughout this paper, we assume that the framework runs the WordCounter application, which counts the appearances of each different word in a set of documents. Table I lists the metrics collected by the monitoring infrastructure when the WordCounter application was running. Monitoring data were collected every 15 seconds and used in analyses to address the questions posed in Section I.

The reference scenario described above leads us to make two observations: first, although we chose SCS as the reference middleware technology, the results presented in the next section can be generalized for any infrastructure presenting equivalent abstractions (application components, containers, and execution nodes); and, second, the time interval in which data were collected can be changed. However, as the value

of 15 seconds was sufficient to capture compliance with and violations of SLOs in the MapReduce application, for the purpose of this work, we kept it as a constant.

III. Experiments

This section focuses on addressing questions $\mathbf{Q_1}$, $\mathbf{Q_2}$ and $\mathbf{Q_3}$. In the course of the investigation, we implemented a management architecture comprising some statistical approaches to deal with each of the three questions. We named this architecture SMART (Self-MAnaged Resource uTilization). In SMART, a local monitor executes in each network node where an application component is instantiated. Each local monitor analyzes the data supplied by the SCS monitoring infrastructure and periodically: constructs performance models of the application, reliably estimates the state of the services (compliance or violation) and, if necessary, diagnoses the cause of a violation. Each individual node is analyzed separately from other nodes. Bellow we describe the approaches we implemented in SMART in order to answer the questions and share the results we obtained. At the end of this section, we present an additional evaluation of the benefits and the overhead imposed by the architecture in a specific application setting.

A. Which statistical learning methods are suitable to forecast performance problems?

In order to answer question $\mathbf{Q_1}$, we cast the performance degradation prediction problem as a supervised classification problem. As described in [4], the performance degradation prediction problem fits naturally into the supervised classification framework. In our experiments, a vector $m \in M$ is a set of values for n collected performance metrics (see Table I) at time t, and label s is one of two states from the set $S = \{s^+ = 1, s^- = 0\}$, where s^+ denotes compliance with an SLO, and s^- represents violation. A log of observations of the metrics collected from the system in operation, a set of examples (m,s), is the training data set supplied for the learning procedure. The learning is supervised because an SLO indicator identifies the values of s corresponding to each observed m. Thus, given an SLO (defined by end-users), an SLO indicator (specified externally), and a training data set (the log of collected metrics), we induce a model of the relationship between M and S. Then, we use the model to decide whether any given set of metric values is more likely to correlate with a compliance with or violation of the SLO. We assume that, in addition to classification, the model provides estimated membership probabilities of the sample in each class. This means that, given a new sample m, the model estimates the probability of m being in compliance ($P(s^+|m)$) with or violation ($P(s^-|m)$) of an SLO[1].

[1]Note that in each training example (m,s), there is an offset l ($l \geq 0$) between the times when m and s are collected, describing how far into the future we wish to predict[12]. For illustrative purpose, in this paper, we considered $l = 0$. Analysis of different values will be explored in future work.

We investigated the following algorithms to build this model: decision tree (DT), Bayesian network (BN), and support vector machine (SVM). These algorithms were chosen because they represent the major families of classifiers in use today. We used the implementations from the Weka [13] library. As these algorithms are well-known in the machine learning literature, we decided to omit their details and, instead, give more information on the used implementations.

Decision tree. We used J48, the classical algorithm for generation of a C4.5 decision tree.

Bayesian network. We focused on restricted network – such as Naive Bayes (NB), Tree-Augmented Naive Bayes (TAN), and Aggregating One-Dependence Estimators (AODE) – and full Bayesian networks (FBN). In the NB algorithm, attributes are assumed to be independent of each other given the class variable. The TAN classifier extends the NB approach by relaxing the independence assumption so that the attributes are connected to each other as a tree. In AODE, an ensemble of TANs are learned and the classification is produced by aggregating the classifications of all qualified TANs. In AODE, a TAN is built for each attribute, in which the attribute variable is set to be the parent of all other attributes. Finally, in FBN, all variables are dependent. In this study, we used the algorithm proposed in [14] to build FBNs. Attributes in this algorithm are assumed to be dependent on each other, and attribute independence is captured in CPTs (conditional probability tables) learned from decision trees. Because the effort on structure learning is reduced by using a full network, this algorithm demonstrates good performance.

Support vector machine. We used WLSVM [15], an implementation of the LibSVM running under Weka environment. We chose LibSVM because it runs much faster than Weka SMO and supports several SVM methods. In particular, we tested SVM with radial kernel, because this kernel can handle the case in which the relation between class labels and attributes is nonlinear, and it requires fewer hyperparameters than other kernels.

Bellow we present a comparative study of the algorithms. After describing the data set used in the evaluation, we draw on the results.

MapReduce application log

We used a data set with $79,319$ instances or samples. It represents a log of executions of the WordCounter application and its behavior (performance) during such executions. The application ran on a cluster of four machines: three running *Workers*, and one running the *Master*. All the machines were interconnected with a 1Gbps Ethernet link. The metrics collected in this log are listed in Table I. During execution, one of the *Worker*s' nodes was submitted to stress with a workload injector. Each execution occurred in a workload scenario that exercised one or more resource types in different intensities. The injection

Metrics related to execution nodes	
net_bytes_in	Number of bytes received from network
net_bytes_out	Number of bytes transmitted to network
disk_bytes_read	Number of bytes read from disk
disk_bytes_write	Number of bytes written to disk
nfs_bytes_read	Number of bytes read from nfs
nfs_bytes_write	Number of bytes written to nfs
cpu_usage	CPU time (system+user)
memory_usage	Amount of memory used
Metrics related to containers	
containers_avg_cpu_usage	CPU usage per container
Metrics related to application components	
avg_response_time	Average response time
map_reduce_phase	Processing phase (map or reduce)

Table I
METRICS COLLECTED AND USED FOR ANALYSIS

	Detection Rate	False Alarm	Balance Accuracy	Training Time
ZR	-	-	34%	2s
NB	57%	13%	74%	3s
TAN	55%	8.8%	79%	19s
AODE	54%	13%	77%	3s
FBN	54%	7%	82%	91s
DT	53%	7%	82%	23s
SVM	54%	7%	83%	6500s

Table II
COMPARING ALGORITHMS AGAINST ACCURACY AND SPEED

comprised three workload patterns: CPU, disk and network. The intensity of each workload pattern was set to 20%, 50% and 80% of utilization. After logging, potential noises or outliers were removed from the data set by constraining the values of attribute $avg\_response\_time$ to lie in the 5th and 95th percentiles of the variable. Then, the class attribute was estimated by defining a threshold value or SLO for variable $avg\_response\_time$. Each sample with $avg\_response\_time$ above the SLO was categorized as a violation. The SLO was computed as a percentile of $avg\_response\_time$. In the experiments carried out in this work, SLO varied from 50 to 90 percentile. After class estimation, the data set was discretized into five levels.

Results

In this experiment, we evaluated the applicability of the six algorithms (DT, NB, TAN, AODE, FBN, and SVM) to derive models to assist autonomic performance management. To this end, we contrasted the algorithms against a variety of characteristics that are important for building online performance models, such as the ones bellow.

- Accuracy: establishes the predictive power of the different methods.
- Time requested for training and classification: establishes the computational times of the algorithms for training and classification.
- Sensitivity to the number of samples in violation: evaluates the algorithm behavior in data sets with a small number of samples in violation. This test is very important in the context of performance problems, since performance degradation is often a rare event.
- Sensitivity to the training data set size: establishes the algorithm behavior in small data sets. This is important to characterize algorithms that present an adequate behavior even when deployed in dynamic environments, where models have to be frequently reconstructed.

In order to establish the predictive power of the algorithms, we used *balanced accuracy*. This metric weights the performance of the models on each of the two classes equally, and, is therefore more informative than straight prediction accuracy in settings in which one of the target classes is rare [4]. The evaluation also included the ZeroR classifier (ZR), which predicts the mode of the labels. This classifier was considered for the purpose of base comparison. Table II shows the performance for each algorithm across the *MapReduce application log* using 10-fold cross-validation. The columns in the table reflect the detection rate for violation, false alarm for violation, balance accuracy, and time required for training attained by the indicated algorithm averaged over the five SLO definitions (50, 60, 70, 80, and 90 percentile of $avg\_response\_time$). The parameter settings for the algorithms were as follows. NB, FBN, and AODE used Weka default parameters. TAN used ENTROPY as the quality score. We also tried BAYES and MDL scores, but we found no significant improvement by changing the quality score. DT and SVM curves were created by performing parameter selection by cross-validation over the parameters of the algorithms. For DT, a pruned tree with confidence factor of 0.3 provided the best result. For SVM, the radial kernel was used with the parameters adjusted as: Capacity $C = 32$ and kernel width $G = 0.5$.

Except for ZR, all classifiers showed a high balance accuracy. ZR, on the contrary, performed badly, showing that most cases yielded non-trivial predictions. Throughout the experiment, SVM showed the best average accuracy. However, the accuracy came at the expense of the highest computational time and many parameters to set. TAN, DT, and FBN also performed well, but the computational time of the latter was higher. NB and AODE generated more false alarms but were, in turn, able to achieve high detection rates and the best computational times. We also evaluated the computational time required for classifying a single sample. For all classifiers, this time was negligible (less than 1 ms), except for SVM, whose classification took 5 ms.

Table II shows that the average accuracy attained by each classifier over the SLO definitions was high. However, the

average detection rate fared worse, because it is more sensitive to the SLO definition than the balance accuracy. Because of that, in order to evaluate the sensitivity of the algorithms in relation to the number of samples in violation, we chose to plot the detection rate, rather than the accuracy, as a function of the SLO definition. Figure 1(a) shows that, as expected, as the SLO increased from 50 to 90 percentile, the detection rate decreased. This happens because the SLO represents a threshold on the average response time of the services. As the percentile increases, so does the threshold, and the number of samples in violation decreases. Computing the slope of each curve in Figure 1(a), we observed that NB and TAN were less sensitive to the reduction of samples in violation than the other classifiers.

We evaluated the classifiers in relation to their sensitivity to the training data set size. We generated new data sets with sizes varying from 10 to 500 samples. Each new data set was generated by randomly picking samples from the original training data set. Then, we computed the accuracy of all classifiers across all new data sets. Figure 1(b) plots the result for this test until the training window reaches 150 samples. We can observe that the accuracy of the classifiers improved over data set size but gradually saturated. In general, starting from 60 samples, the accuracy of the classifiers was satisfactory. Before that, for smaller data sets, SVM performed better than the other algorithms. AODE and NB presented the worst performance for small data sets.

Overall Evaluation

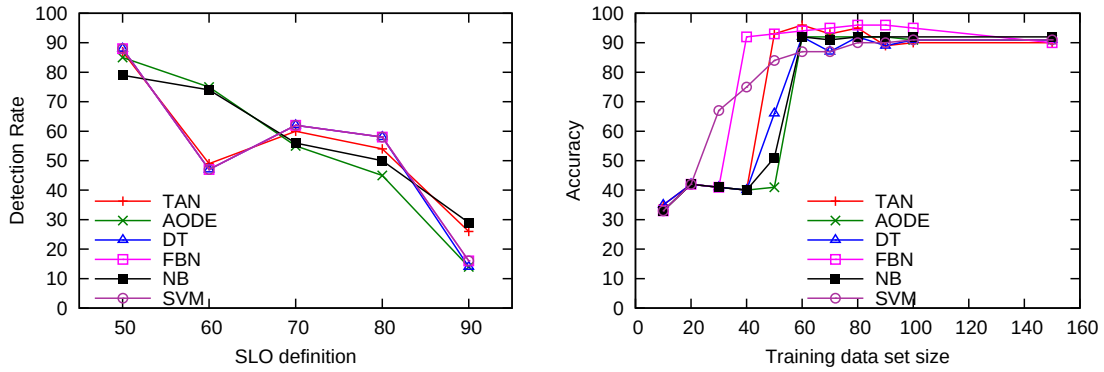
Although all six classifiers performed comparably well in terms of accuracy, the SVM provided the best overall results, even for small data sets. However, the time required to train the SVM classifier was very expensive. In addition to this cost, the time required to search the parameter values to find the optimum performance also has to be taken account. For example, in our experiments, it took almost one hour for performing such search. Therefore, we considered the SVM classifier learned here prohibitive to derive online performance models, especially in dynamic environments where the models have to be frequently reconstructed. The DT and FBN classifiers provided the second-best accuracy and the same false alarm rate provided by SVM. Indeed, both classifiers behaved very similarly in terms of accuracy. However, while FBN performed well in small data sets, DT performed more poorly than expected. FBN presented the second-worst computational time. Because of these disadvantages, we considered these methods less suitable for autonomic performance management. On the other hand, the NB, AODE and TAN classifiers, although less successful than the other classifiers in terms of accuracy and false alarm rate, were considerably better in terms of computational time. In fact, these algorithms presented the best trade-off between accuracy and computational time. In addition, these models were also less sensitive to the number of samples in violation available in the training data set. Particularly, we considered TAN and AODE the winners, since they outperformed NB while they were also computational simple. TAN had a number of advantages over the other classifiers: fast training time, high accuracy and little sensitivity to both, small training data sets and few samples in violation. AODE, in turn, is inherently incremental, as the algorithm makes no model selection. We highlight that, for all classifiers, it was difficult to keep the false alarm rate low enough. Indeed, as argued by Powers et al. in [5], classification methods generate high false alarm rates. In the next section, we will address this problem.

B. Can we make the prediction process more robust against failures of the learning method?

Considering the performance degradation prediction problem, a typical training data set can be obtained using standard synthetic benchmarks or data sets coming from real or production environments. In general, synthetic benchmarks are not sufficiently rich to produce the wide range of system conditions that might occur in practice [4], and, the resulting training data set consequently exhibits statistical variations in relation to real conditions. On the other hand, data sets coming from real or production environments are richer but subject to noise data which manifest randomly. Therefore, when machine learning algorithms are deployed in production settings, they are subject to transient failures, whether they were trained using benchmarks or real data sets. These failures can produce an unacceptable level of false alarms and affect the confidence on the management solution.

A preliminary examination of the classifier results was done by plotting them as a function of time, when the system is submitted to an increasing CPU workload. Figure 2 shows the behavior of the workload and the probability of violation ($P(s^-|m)$) outputted by a TAN classifier in a trace of approximately 75-minute run. Each point in the graphs represents the workload intensity (Figure 2(a)) or the probability of violation (Figure 2(b)) averaged over ten samples, each one collected at 15-second intervals. Two observations can be made from Figure 2. Firstly, the probability of violation increased as the workload intensity increased. Thus, a pattern of increasing probability of violation is indicative of performance problems. Secondly, even though the sequence of values of $P(s^-|m)$ was an increasing trend curve, it was not monotonic since the classifier was subject to failures. These observations led us to investigate statistical tests for trend as a method for monitoring the outcomes provided by classifiers.

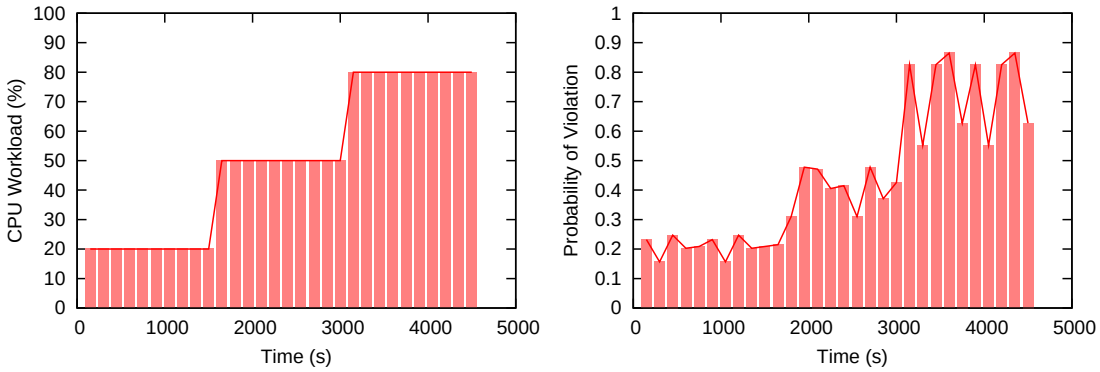
We used a fast and very simple nonparametric statistical test, the *reverse arrangements test* [16] (or RAG test), to monitor p_i, $i = 1...N$, a time sequence of the most recent values predicted for $P(s^-|m)$. Given this time sequence, the test computes $A = \sum_{i=1}^{N-1} A_i$, the sum of all reverse



(a) Detection rate in relation to SLO definition

(b) Accuracy in relation to training data set size

Figure 1. Comparing accuracy against SLO definition and training data set size



(a) Workload behavior

(b) Probability of violation behavior

Figure 2. Workload and probability of violation in a system subject to stress

arrangements, being a reverse arrangement defined as an occurrence of $p_i > p_j$ when $i < j$. Given a significance level α, the hypothesis of no trend in the sequence p_i is given by the interval $A_{N;1-\alpha/2} \leq A \leq A_{N;\alpha/2}$. This means that, if the sum of all reverse arrangements, A, does not fall into the interval, the hypothesis of trend in the sequence p_i is accepted. If A falls under the lower limit, the test indicates an increasing trend. Conversely, if A falls above the upper limit, the test marks a decreasing trend.

In addition to the statistical test for trend described above, we also applied a smoothing method, the *weighted moving average* or WMA approach [16], to reduce the effect produced by random variations in the sequence p_i. Given the sequence, we compute WMA as $\frac{\sum_{i=1}^{N} i*p_i}{\sum_{i=1}^{N} i}$.

Algorithm 1 represents the warning algorithm resulting from the combination of both approaches. We keep a sequence of the last N probabilities of violation outputted by the classifier. Every time a new probability is available, it replaces the oldest one in the sequence, triggering the computation of RAG test and WMA. The algorithm raises an alarm when WMA reaches $thUpper$ or the RAG test detects increasing trend at a certain probability of violation value ($thRag$). On the other hand, the alarm is disabled when WMA decreases to $thLower$ or RAG test detects decreasing trend at a certain probability of violation value. The thresholds and the significance level α form the parameters that control the trade-off between robustness and fast reaction to performance degradation.

Results

In this experiment we used two workload patterns with high oscillations, proposed in [4], to evaluate the effectiveness of the warning algorithm. Firstly, we submitted the system running the WordCounter application to a CPU workload resembling a sinusoid overlaid under a ramp. This test was an approximately 2-hour run, in which the CPU load moved back and forth steadily. The intent of this workload was to mimic sudden and short burst of increasing workload. The TAN classifier was used in this test and a violation was defined as occurring when cpu_usage exceeds 50%. The warning algorithm was configured as: $thUpper = 0.5$, $thLower = 0.2$, $thRag = 0.4$, $N = 10$ and $\alpha = 0.01$.

Algorithm 1 Combining RAG test and WMA to build a warning algorithm

Require: A time sequence p_i of the most recent values predicted for the probability of violation. Thresholds $thUpper$, $thLower$ and $thRag$. Significance level α

Ensure: alarm_on:0 (off) or alarm_on:1 (on)

1: Given p_i, $i = 1...N$, compute A and WMA
2: **if** $WMA \geq thUpper$ or ($p_N \geq thRag$ and $A < A_{N;1-\alpha/2}$) **then**
3: alarm_on = 1;
4: **else if** $WMA \leq thLower$ or ($p_N \leq thRag$ and $A > A_{N;\alpha/2}$) **then**
5: alarm_on = 0;
6: **end if**

Figure 3(a) plots: *i*) the CPU workload, *ii*) the false alarms (for violation) emitted by the classifier, *iii*) the moment the warning algorithm raised the alarm, and *iv*) the SLO threshold value – as a function of time. The plot shows that, while the workload oscillated rapidly, the immediate value of the probability also oscillated and false alarms were outputted. Throughout all of the test, however, the warning algorithm was able to catch the prevalent trend with no false alarms. Figure 3(a) shows that the alarm was raised only few seconds after the CPU utilization reaches or exceeds the threshold value.

Secondly, using the same configurations defined in the sinusoid workload test, we submitted the system running the WordCounter application to a CPU workload resembling a step. This was a 2.5-hour run of an on/off workload, in which the CPU load consisted of 12-minute long bursts with 12 minutes between bursts. The intent of this workload was to mimic sudden but sustained bursts of increasingly intense workload against a backdrop of moderate activity. Figure 3(b) plots the result of this test. We see that, although the warning algorithm did not issue any false alarm, it was not able to keep the alarm raised during the time interval in which the CPU utilization reached 50%. The cause of this misbehavior is explained by the classifier, which performed poorly during this interval. As the CPU utilization reached 80%, however, the warning algorithm performed as expected. These results suggest that the warning algorithm helps to improve the resilience to transient failures of the classifiers. However, the effectiveness of the algorithm is affected by successive errors of the classifiers.

C. Can we quantify the influence of each monitored metric on a given performance problem?

Although some classification algorithms, such as Bayesian networks and decision trees, answer questions about which attributes and variables are the most important in a decision, many classification methods, like neural networks and SVM, do not have such interpretability properties. Motivated by this limitation, we investigated machine learning independent alternatives to the problem. In this subsection, we introduce three statistical tests we applied to answer question **Q$_3$**: *reverse arrangements test*, *z-scores* and *chi-square*. Typically, the first two tests attempt to learn the difference between two populations or, in our case, the population in compliance with an SLO, and the population violating an SLO. The chi-square test, in turn, is applied to estimate the independence of two variables, that is, to test the hypothesis that two attributes (in our case, an ordinary attribute and the class variable) are independent. We assumed that the diagnostic algorithms are triggered when the warning algorithm raises an alarm for performance degradation and that the most recent samples used for forecasting performance problems are kept in memory. We denoted these samples as *warning* data set. We also assumed that a *reference* data set, taken from the compliance population, is kept in memory. The reference data never change, but the warning data does since it is the last samples in use in the warning algorithm.

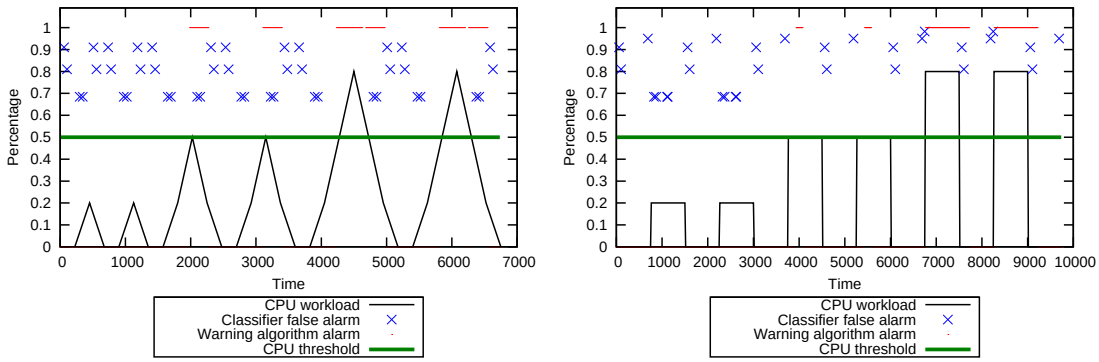
Reverse arrangements test. To apply the reverse arrangements test for the purpose of diagnosis, we use the warning data set and compare it to the reference data. For each attribute, we apply the test for both data sets: warning and reference. Based on the assumption that a pattern of increasing attribute values is indicative of performance degradation, the attributes which correlate more with a violation condition are those that present significant increasing trend for the warning data set and no trend for the reference data. Then, we rank these attributes (candidate attributes) by their number of reverse arrangements computed using the warning data set. The lower the number, the greater the degree of correlation with a violation condition is.

Z-scores test. In the z-scores test, the warning data set is also compared to the reference data. For each attribute, we compute z as:

$$z = \frac{m_v - m_c}{\sqrt{\frac{\sigma_v^2}{n_v} + \frac{\sigma_c^2}{n_c}}} \quad (1)$$

In Equation 1, m_v, σ_v^2 are the mean and variance of the attribute in the warning data, m_c and σ_c^2 are the mean and variance of the attribute in the reference data, n_v is the number of samples in the warning data set and n_c is the number of samples in the reference data. The z-scores test compares the mean values of each attribute in both data sets. Large positive z-scores indicate that the attribute is higher in the population of violation.

Chi-square test. In order to use the chi-square test for the purpose of diagnosis, we combine the warning and reference data and apply the test to the resulting data set. The chi-square algorithm [16] evaluates attributes individually by measuring the chi-square statistic with respect to the class variable. Using a contingency table, i.e., a frequency table where a sample is classified according to two different



(a) Robustness test for sinusoid workload

(b) Robustness test for step workload

Figure 3. Testing the robustness against false alarms in the warning algorithm

attributes, the chi-square statistic is computed as:

$$\chi^2 = \sum_{i,j} \frac{(f_{ij} - N \times \pi_{ij})^2}{N \times \pi_{ij}} \quad (2)$$

In Equation 2, f_{ij} and $N \times \pi_{ij}$ are, respectively, the observed and expected frequencies in the i, j^{th} cell of the contingency table. N is the number of samples and π_{ij} is the probability, in the population, that an individual selected at random will fall into attribute values x_i and y_j. The greater the chi-square value, the greater the degree of correlation is between the attribute and the class.

Results

We evaluated the accuracy of the statistical tests for diagnosing metrics related to performance degradation. We tested the methods using data collected from the execution of the WordCounter application in different workload scenarios. Altogether, we created five data sets, each representing 12.5 minutes of system data, taken at 15 second intervals. Each data set was created from a workload scenario comprising three consecutive workload components, $\langle c_1, c_2, c_3 \rangle$, with durations of 300, 225, and 225 seconds, respectively. Each workload component exercises one or more resource types at different intensities. A workload component c_k=i_c:i_d:i_n, $k = 1, 2, 3$, denotes a pattern of i_c CPU, i_d disk and i_n net workload intensity. The five data sets and the workload scenarios from which they were derived are shown in the first five lines of Table III. The reference data comprised 50 random samples taken from a log of execution of the WordCounter application with no stress applied to the system. The warning data sets were taken to be the 50 samples of each data set. Additionally, we tested the methods using the warning data set available when the warning algorithm (described in subsection III-B) raises an alarm. This data set was collected from the *step* workload experiment (see Figure 3(b)) and consisted of the last 50 samples collected before the alarm being activated.

In order to evaluate the accuracy of the proposed tests, we defined two sets: $Rank$, the set of the first three metrics ranked by an algorithm, and $Stress$, the set of resources (i.e. CPU, disk, etc) stressed in a given experiment. We defined $S_{rank} = \sum_{i=1}^{3}(4-i) * h_i$ as the score an algorithm accumulates when diagnosing the causes of a performance degradation. In this case, the term $(4 - i)$ is the reward associated to the i^{th} metric of $Rank$, and h_i is an indicator function that evaluates to 0 if the i^{th} metric of $Rank$ is not related to any resource of $Stress$; otherwise, the function evaluates to 1.

We evaluated each method across the six data sets (listed in Table III). Table IV demonstrates the results for *disk*, *cpu-net* and *cpu-step* data sets. For *disk* data set, for example, we see that reverse arrangements incorrectly ranked $memory\_usage$ as the metric most related to violations. The second metric, $disk\_bytes\_read$ is closely related to disk usage, but the third metric is not. Thus, considering *disk* data set, the S_{rank} score for reverse arrangements was $3 \times 0 + 2 + 1 \times 0 = 2$. This score was lower than the scores of the other two algorithms, as the first two metrics selected by these tests are closely related to disk usage. In general, the chi-square test exhibited the best scores across the first five data sets, followed by z-scores and, then, reverse arrangements. However, the attempt at diagnosing in the *cpu-step* data set painted a quite different picture, and the chi-square test fared worse. The cause for this poor performance is the small number of samples in violation in *cpu-step* data set. Since this data set comprises the last samples before the alarm being raised, there is a preponderance of non-violation in most samples. z-scores and reverse arrangements, in turn, are not sensitive to the number of samples in violation, since they quantify trends in time sequences. Nonetheless, the reverse arrangements test performed more poorly than z-scores. We believe that the time-order information required by reverse arrangements was not important for diagnosing performance problem. To

confirm this suspicion, we will investigate other statistical tests which make no assumption about time-order. We also evaluated the computational time of the algorithms. Reverse arrangements, z-scores and chi-square took, respectively, 231, 415 and 425 seconds. These numbers represent the average time needed for each algorithm to perform a diagnosis in each of the six data sets when the reference and the warning data set each has 50 samples. All the three tests took less than one second to perform diagnosis, confirming the efficiency of statistical tests. These experiments support the conclusion that z-scores is more appropriate to our diagnostic engine.

D. Additional remarks on experiments

In addition to the previous experiments, we evaluated the potential benefits of SMART in capturing the performance behavior of component-based applications. In this experiment, the MapReduce framework employed the inferred performance information provided by SMART to make scheduling decisions and improve the overall application execution time. To this end, we endowed the framework with two scheduling policies: CPU-based, and SMART-based. In the first policy, a pool of idle *Workers* is sorted by the increasing values of the CPU utilization of the nodes. In the second policy, the pool is sorted by the increasing values of the violation probability. We used five nodes from the cluster described in Subsection III-A: one running the *Master*, and four running *Workers*. One of the *Workers* nodes was submitted to a workload pattern of 50% of CPU and 50% of disk. TAN was defined as the classifier, and the SLO was set to 80 percentile of $avg\_response\_time$. In these conditions, the application running with the SMART-based policy took 183.80 seconds to complete, while the application running with the CPU-based policy took 207.80 seconds. Each execution time is the average of ten executions. The performance of the application running with the SMART-based policy was approximately 11% better than the other application. This result provided empirical support for concluding that, even relatively simple component-based applications, such as MapReduce applications, are too complex for being modeled by simple rules, such as the CPU utilization. The analysis engine provided by SMART, in turn, is more suitable to capture the behavior of such applications.

We also evaluated the overhead imposed by SMART. Using the same nodes and parameter settings of the previous experiment, except for the workload injection, we evaluated the performance of the MapReduce application in three settings: *i*) monitoring and analysis services disabled, *ii*) only the monitoring service enabled, and *iii*) monitoring and analysis services (SMART) enabled . It took 139.44, 142.06 and 145.52 seconds to execute the application in settings *i*, *ii*, and *iii*, respectively. Each execution time is the average of ten executions. We observe that the overhead imposed by the monitoring service over the MapReduce framework was approximately 2%, and increased to 4% when the analysis service was considered. Therefore, SMART imposed no expressive performance loss to applications.

IV. Related Work

Much work has been done on performance characterization of distributed systems, and there are two principal schools in the literature. The first school consists of approaches concerning analytical models on network queue [2], [1] or control theory [3] to model the application behavior. Analytical models have the following disadvantages: they require substantial effort from human experts and may be subject to mistakes or unrealistic assumptions; and, because they are difficult to derive, most models limit to describe the application behavior in terms of CPU utilization.

The second school consists of approaches applying data mining and machine learning techniques to induce performance models from instrumentation data. Unlike those in the first school, theses approaches assume little domain knowledge and can be programmatically updated. Probabilistic and machine learning-based models have been successfully used in forecasting system resource utilizations and performance. Among these works, Andrzejak et al. [12] use machine learning methods to predict performance degradation caused by software aging, and Sahoo et al. [17] apply time series and Bayesian network to predict system utilization. Statistical learning techniques have also been applied successfully in diagnostic tasks, such as performance debug [6] and attributing performance problems to low-level system features [4], among others. There have been few works concentrating on evaluating the fit and applicability of different statistical learning techniques to deal with performance problems. A comparative study of Bayesian network and neural networks for modeling response time in service-oriented systems is presented by Zhang et al. in [8]. However, only Bayesian network and neural networks are considered in the study and the effect of different instantiations and parametrizations is not explored in the work. A more detailed work is presented by Powers et al. in [5], in which the authors compare the performance of regression methods and Bayesian network classifiers to forecast performance problems. Our work differs from the above in various way. We analyze the performance problem under three dimensions: prediction, robustness and diagnosis. Under the prediction dimension, the major families of classifiers are investigated through a wide range of characteristics. Under the diagnosis dimension we propose the use of statistical tests and we make the diagnosis problem independent of the classifier used in the prediction process. Finally, none of the above works deal with the robustness dimension.

V. Conclusion

In this paper we have addressed the characterization of performance problems under three dimensions. First, we

Data Set	Scenario
cpu	⟨20:00:00,50:00:00,80:00:00⟩
disk	⟨00:20:00,00:50:00,00:80:00⟩
net	⟨00:00:20,00:00:50,00:00:80⟩
cpu-disk	⟨20:20:00,50:50:00,50:80:00⟩
cpu-net	⟨20:00:20,50:00:50,50:00:80⟩
cpu-step	*step* workload experiment

Table III
DATA SETS FOR DIAGNOSTIC EVALUATION

Data Set	reverse arrangements		z-scores		chi-square	
	Attributes	S_{rank}	Attributes	S_{rank}	Attributes	S_{rank}
disk	memory_usage disk_bytes_read net_bytes_in	2	disk_bytes_write disk_bytes_read nfs_bytes_write	5	disk_bytes_read disk_bytes_write net_bytes_in	5
cpu-net	memory_usage cpu_usage net_bytes_in	3	net_bytes_out cpu_usage disk_bytes_read	5	net_bytes_in cpu_usage net_bytes_out	6
cpu-step	net_bytes_in net_bytes_out nfs_bytes_read	0	cpu_usage containers_avg net_bytes_in	5	disk_bytes_write nfs_bytes_write cpu_usage	1

Table IV
METRICS SELECTED BY THE STATISTICAL TESTS

examined the abilities of the major families of classifiers in capturing the performance behavior of middleware based-systems. The evaluation showed that most of the characteristics considered in the study favor Bayesian networks, especially TAN and AODE. The evaluation also confirmed that classification methods generate high false alarm rates. Motivated by this fact, we added another dimension to the problem: robustness. We presented a warning algorithm that has proven to be effective in augmenting the robustness of learning methods against false alarms. Finally, we analyzed the problem under the diagnosis dimension. Our study showed the applicability and efficiency of statistical tests, especially the z-scores test, for characterizing the potential change in metric trends automatically. Although the presented results were obtained from experiments with a specific application on top of a specific component-based middleware, we believe these results can be generalized to other applications and middleware technologies. Future work includes evaluating the approaches presented in this paper along with other types of systems.

REFERENCES

[1] S. Chen, Y. Liu, I. Gorton, and A. Liu, "Performance prediction of component-based applications," *J. Syst. Softw.*, vol. 74, no. 1, pp. 35–43, 2005.

[2] Q. Zhang, L. Cherkasova, N. Mi, and E. Smirni, "A regression-based analytic model for capacity planning of multi-tier applications," *Cluster Computing*, vol. 11, no. 3, pp. 197–211, 2008.

[3] D. Kusic and N. Kandasamy, "Risk-aware limited lookahead control for dynamic resource provisioning in enterprise computing systems," *Cluster Computing*, vol. 10, no. 4, pp. 395–408, 2007.

[4] I. Cohen, M. Goldszmidt, T. Kelly, J. Symons, and J. Chase, "Correlating instrumentation data to system states: a building block for automated diagnosis and control," in *Proceedings of OSDI 2004*. USENIX Association, 2004, pp. 231–244.

[5] R. Powers, M. Goldszmidt, and I. Cohen, "Short term performance forecasting in enterprise systems," in *Proceedings of KDD '05*. ACM, 2005, pp. 801–807.

[6] P. Barham, A. Donnelly, R. Isaacs, and R. Mortier, "Using Magpie for request extraction and workload modelling," in *Proceedings of OSDI 2004*. Berkeley, USA: USENIX Association, 2004, pp. 259–272.

[7] R. Kohavi, "Scaling up the accuracy of nave-bayes classifiers: A decision-tree hybrid," in *Proceedings of the Second International Conference on Knowledge Discovery and Data Mining*. AAAI Press, 1996, pp. 202–207.

[8] R. Zhang and A. J. Bivens, "Comparing the use of bayesian networks and neural networks in response time modeling for service-oriented systems," in *Proceedings of SOCP '07*. ACM, 2007, pp. 67–74.

[9] Tecgraf/PUC-Rio, "SCS - software component system." [Online]. Available: http://www.tecgraf.puc-rio.br/scs/

[10] E. Fonseca, S. Correa, and R. Cerqueira, "Experimenting middleware-level monitoring facilities to observe component-based applications," in *II Brazilian Symp. on Software Components, Architectures, and Reuse*. EDIPUCRS, pp. 96–106.

[11] J. Dean and S. Ghemawat, "Mapreduce: Simplified data processing on large clusters," in *Proceedings of OSDI 2004*. USENIX Association, 2004, pp. 137–150.

[12] A. Andrzejak and L. M. Silva, "Using machine learning for non-intrusive modeling and prediction of software aging," in *NOMS*. IEEE, 2008, pp. 25–32.

[13] I. H. Witten and E. Frank, *Data Mining: Practical Machine Learning Tools and Techniques*. Morgan Kaufmann, 2005.

[14] J. Su and H. Zhang, "Full bayesian network classifiers," in *ICML '06: Proceedings of the 23rd international conference on Machine learning*. ACM, 2006, pp. 897–904.

[15] Y. EL-Manzalawy and V. Honavar, *WLSVM: Integrating LibSVM into Weka Environment*, 2005, software available at http://www.cs.iastate.edu/~yasser/wlsvm.

[16] NIST/SEMATECH, "e-handbook of statistical methods." [Online]. Available: http://www.itl.nist.gov/div898/handbook/

[17] R. K. Sahoo, A. J. Oliner, I. Rish, M. Gupta, J. E. Moreira, S. Ma, R. Vilalta, and A. Sivasubramaniam, "Critical event prediction for proactive management in large-scale computer clusters," in *Proc. of KDD '03*. ACM, 2003, pp. 426–435.