Ranking the Importance of Alerts for Problem Determination in Large Computer Systems

Guofei Jiang
NEC Laboratories America
Princeton, NJ 08540
gfj@nec-labs.com

Haifeng Chen
NEC Laboratories America
Princeton, NJ 08540
haifeng@nec-labs.com

Kenji Yoshihira
NEC Laboratories America
Princeton, NJ 08540
kenji@nec-labs.com

Akhilesh Saxena
NEC Laboratories America
Princeton, NJ 08540
saxena@nec-labs.com

ABSTRACT

The complexity of large computer systems has raised unprecedented challenges for system management. In practice, operators often collect large volume of monitoring data from system components and set up many rules to check data and trigger alerts. However, the alerts from various rules usually have different problem reporting accuracy because their thresholds are often manually set based on operators' experience and intuition. Meantime, due to system dependencies, a single problem may trigger many alerts at the same time in large systems and the critical question is which alert should be analyzed first in the following problem determination process. In this paper, we propose a novel peer review mechanism to rank the importance of alerts and the top ranked alerts are more likely to be true positives. After comparing a metric value against its threshold to generate alerts, we also compare the value with the equivalent thresholds from many other rules to determine the importance of alerts. Our approach is evaluated with a real test bed system and experimental results are also included to demonstrate its effectiveness.

Categories and Subject Descriptors

K.6 [**Management of Computing and Information Systems**]: System Management

General Terms

Management

Keywords

Fault management, rule management, alert ranking, fault model, invariant network, peer review

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ICAC'09, June 15–19, 2009, Barcelona, Spain.
Copyright 2009 ACM 978-1-60558-564-2/09/06 ...$5.00.

1. INTRODUCTION

With the popularity of online services, many large-scale distributed systems and data centers have been built as the critical infrastructures to accommodate millions of online users simultaneously. The complexity of these giant systems has also raised unprecedented challenges for operators to maintain and manage them. These systems are usually deployed by integrating thousands of heterogeneous components including servers, routers, storage devices and software, which are typically provided by different vendors. Compared to large hardware-based systems such as telephone systems, the numerous software components running on computing systems obscure the dependencies and interactions among system components. While individual components like operating systems are already complex enough, the massive number of implicit component dependencies introduced by software have dramatically increased the complexity of today's computing systems. Meantime, many of such large systems are not static but always evolving with numerous changes such as security patch installations, software or hardware upgrades and configuration modifications. Therefore, the system scale, heterogeneity and dynamics as well as hidden dependencies all contribute to the difficulties in complexity management.

Many of such large systems are essentially mission critical systems and even minutes of system down time could lead to big revenue loss in business and further affect our normal life [17]. For example, a system failure of BlackBerry email service affected millions of customers on April 17, 2007 [3]. Amazon's storage service S3 was down for several hours on February 15, 2008, which affected its customers worldwide [2]. Therefore, service providers usually setup a large system management team to operate their infrastructures and services. In practice, operators collect large volume of monitoring data from system components to track the status of their infrastructures. Since it is impossible to manually scan and interpret large volume of data in real time, operators usually set many rules to check data and trigger alerts. For example, if a selected metric (e.g. CPU usage) exceeds a given threshold, an alert is generated to notify the operators who might follow up with an examination.

However, due to system complexity, it is difficult to set good thresholds in rules and a bad threshold often leads to large false positives or false negatives in problem reporting. This is especially a problem for 24*365 operational

systems such as online services. Meantime, different operators may set different rules and thresholds in their systems based on their personal management experiences and intuitions. For example, compared to a novice, an experienced operator may set better rules and thresholds; some operators might always tend to set higher thresholds than others. As a result, the alerts originating from various rules have different accuracy in problem reporting and some alerts may be more "important" than others. In fact, it is difficult to normalize various rules and thresholds in rule management, especially for large systems that are managed by many operators. Meantime, due to system dependencies, a single fault or performance problem may trigger many alerts at the same time. Now the critical question is which alert should be analyzed first because analyzing all alerts might take too much time to fix the problem and obviously not all of alerts are same important. For mission critical systems, it is very important to reduce MTTR(Mean Time to Recovery) so as to keep high system reliability and availability [15].

Prior approaches use problem signatures to correlate a set of alerts [20], i.e., a specific set of alerts are considered as the signature of a known problem. In large systems, many problems are not anticipated and well understood. Due to system dynamics and uncertainties, even the same problem may manifest itself in different ways. Meantime, many large systems are customized for specific services and also keep evolving along time. Therefore, it is very difficult to precisely define problem signatures and further use them for problem determination. In this paper, we propose a novel peer-review mechanism to rank the importance of alerts so that operators can consult the ranking to prioritize their problem determination process. Essentially we blend the system management knowledge from multiple operators by using intelligent data analysis. More specifically, we have made the following contributions:

- We propose a value propagation mechanism to map the threshold of one rule into its equivalent values under other rules. In our previous work [9] [11], we proposed data mining algorithms to automatically extract many invariant relationships among system monitoring data. In this paper, such an invariant relationship enables us to compare the threshold of one metric with that of another metric.

- We introduce a collaborative peer review mechanism so that a measurement is not only checked by its own rule but also other rules. As a result, an alert is ranked as more important if it gets more consensus from other rules. In current rule-based systems, each rule works in its isolated local context by comparing a measurement with its threshold. In this paper, such a measurement is further compared with the equivalent thresholds mapped from other rules to rank its alerts in a global context.

- We also evaluate our approach with a real test bed system and experimental results are also included to demonstrate its effectiveness.

2. RELATED WORK

During the last several years, we have focused on developing invariant-based frameworks for system management. To the best of our knowledge, we are the first research group that proposed the concept of system invariants and verified their existence in large computing systems [9] [11]. In our previous work [10] [7], we also developed linear and probabilistic models to correlate monitoring data for fault detection and isolation in complex systems. However, none of our previous works is involved with rule-based alert management, which is the focus of this paper. Though our approach might appear simplistic, it does address a very important problem in system management practice.

In practice, rule-based systems are widely deployed for system management [19] [14]. However, in current systems, each measurement is usually compared with its own threshold to generate alerts and each rule works in its isolated local context. Several solutions have been proposed for event correlation and alarm filtering with applications in network management. Yemini et.al. [20] proposed a "codebook" approach to correlate each known problem with a specific set of alerts/events, which is essentially used as the signature in problem determination. Gruschke [6] employed dependency graphs to represent correlation knowledge (i.e., problem signature) and further proposed an integrated event management approach. Based on the prior knowledge of network dependency, there also exists some work on alarm filtering [13]. However, these approaches assume that we have the prior knowledge of system dependency and problem signatures. As discussed earlier, it is difficult to extract system dependencies or define problem signatures in complex computer systems because of their massive software dependencies. In this paper, we do not directly correlate a set of alerts with a problem. Instead, we rank the importance of alerts by introducing a peer-review mechanism so that operators can consult the ranking to prioritize their follow-up examinations.

Prior work has focused on problem determination in distributed systems though these works are not related to rule management. Cohen et.al. [5] used a tree-augmented naive (TAN) bayesian network to learn the probabilistic relationship between SLA violations and resource usages. They further used this learned bayesian network to identify performance bottlenecks during SLA violations. In their Elba project, Parekh et.al. [16] compared the TAN bayesian network with other classifiers like decision trees in performance bottle detection. Chen et.al. [4] modified the JBoss middleware to trace user requests in the J2EE platform, and developed two methods to use collected traces for fault detection and diagnosis. Aguilera et.al. [1] proposed two algorithms to isolate performance bottlenecks in distributed systems composed of black-box nodes. Our paper focuses on how to rank the alerts from existing rule-based systems for problem determination so that we can not compare our work with these problem determination solutions.

3. SYSTEM INVARIANTS

Before we propose our rule management approach, in this section we need to introduce an important concept named system invariants, which characterize the hidden invariant relationships among system monitoring metrics. As discussed earlier, we use such an invariant relationship to map the threshold of one metric into an equivalent threshold in another metric, which essentially enables us to rank the thresholds from various rules and determine the importance of alerts.

Operators collect large amount of monitoring data from

complex systems to track their operational status. Log files and network traffic statistics are typical examples of such monitoring data. This monitoring data can be considered as the observable of internal system state. For online services, when large volume of user requests flow through distributed systems, many of internal measurements respond to the volume of workloads accordingly. For example, network traffic volume and CPU usage are driven to go up and down by the intensity of workloads. In this paper we use a general term named flow intensity to describe the intensity with which internal measurements respond to the volume of workloads. For example, number of SQL queries and average CPU usage (per sampling unit) are such flow intensity measurements. For convenience, we use variables like x and y to represent flow intensity measurements.

Since flow intensity measurements are mainly driven to change by the same external factor - the intensity of workloads, they have similar evolving curves along time t. As time series, many flow intensity measurements have strong correlations and here we use equations like $y = f(x)$ to characterize the relationship between two measurements x and y. If such relationships always hold along time, they are considered as the invariants of the underlying system. No matter how workloads change, such system invariants remain to be the same. Note that the equation $y = f(x)$ but not the measurements x and y is considered as an invariant.

With flow intensities measured at various points across large systems, we need to consider how to extract their relationships, i.e., with measurements x and y, how to learn a function f so that we can have $y = f(x)$? In this paper, we use AutoRegressive models with eXogenous inputs (ARX) [12] to learn their linear relationships. At time t, we denote the flow intensities measured at two points by $x(t)$ and $y(t)$ respectively. The ARX model describes the following relationship between two flow intensities:

$$\begin{aligned} & y(t) + a_1 y(t-1) + \cdots + a_n y(t-n) \\ = \; & b_0 x(t-k) + \cdots + b_{m-1} x(t-k-m+1) + b \end{aligned} \quad (1)$$

where $[n, m, k]$ is the order of the model and it determines how many previous steps are affecting the current output. a_i and b_j are the coefficient parameters that reflect how strongly a previous step is affecting the current output. For convenience, we use θ to denote the set of coefficient parameters, i.e., $\theta = [a_1, \cdots, a_n, b_0, \cdots, b_{m-1}, b]^T$. Since there exist time delays in correlating measurements across distributed systems and various system components may also have unsynchronized time clocks, we consider the temporal dependency in the above ARX model. In fact, even with synchronized time clocks, different components may log their data timestamps with various time delays.

Given a window of monitoring data $\{x(t), y(t)\}$,$1 \leq t \leq N$, the Least Squares Method (LSM) is used to find the best θ that minimizes the error between the learned model and the given monitoring data. Our previous work [9] [11] includes the details on how to calculate θ. In this paper, we use the following equation to calculate a normalized fitness score for model validation:

$$F(\theta) = [1 - \sqrt{\frac{\sum_{t=1}^{N} |y(t) - \hat{y}(t|\theta)|^2}{\sum_{t=1}^{N} |y(t) - \bar{y}|^2}}], \quad (2)$$

where $\bar{y}$ is the mean of the real monitoring data $y(t)$. Given the monitoring data $x(t)$ and θ, $\hat{y}(t|\theta)$ is the output from the model shown in Equation (1). Basically Equation (2) introduces a metric to evaluate how well the learned model approximates the real data. Given two flow intensities, we can always learn a model but only a model with high fitness score characterizes their actual relationship. We can set a range of the order $[n, m, k]$ rather than a fixed number to learn a list of model candidates and then select the model with the highest fitness score. Since computer systems respond to workloads quickly, the range of $[n, m, k]$ is usually small. For example, we set $[n, m, k] \leq 2$ in our experiments.

After we learn a model for two flow intensities, we still need to verify whether such a relationship can hold along time. In order to extract invariants from monitoring data automatically, we try any combination of two measurements to construct a model first and then continue to validate whether this model fits with new observations, i.e., we use brute-force search to construct all hypotheses of invariants first and then sequentially test the validity of these hypotheses in operation. Note that we always have sufficient monitoring data from a 24*365 operational system to validate these hypotheses along time. For every time window of monitoring data, we use Equation (2) to calculate the fitness score $F(\theta)$. Since the models with low fitness scores do not characterize the real data relationships, we choose a threshold $\widetilde{F}$ to filter out those models in sequential testing. Therefore, at every time window, a model with fitness score lower than $\widetilde{F}$ will be removed from the following testing process. After many time windows, the remaining stable set of models are considered as the system invariants.

Note that our invariant extraction algorithms are fully automatic and more details can be found in [9]. We verified that such invariants widely exist in large distributed systems, which are governed by the physical properties or software logic constraints of system components. Instead of the brute-force invariant search algorithm, we also proposed efficient and scalable algorithms to extract invariants by trading off search accuracy. The computational efficiency and scalability issues of these algorithms are extensively analyzed in our previous work [11]. As an example, with a common desktop (Intel Dual Core 2.66 GHz CPU, 4MB cache and 2GB memory), even our brute-force search algorithm can extract all invariants from 3000 monitoring metrics within half an hour.

4. VALUE PROPAGATION

In the above section, we introduced the concept of system invariants and our approach on how to automatically model and extract invariants from monitoring data. Figure 1 illustrates a small example of typical invariant networks that profile the relationships of measurements. In this figure, each node i represents a measurement I_i while each edge represents an invariant relationship between the two associated measurements. Since we use a threshold $\widetilde{F}$ to filter out those models with low fitness scores, not any pair of measurements would have invariant relationships. Therefore, there also exist disconnected subnetworks in Figure 1. In practice, since most alerts are set to monitor system resource usages which are directly driven up and down by the volume of workloads, we observe invariant relationships among the majority of monitoring metrics. All edges are bi-directional because we always construct two models (with reverse input and output in Equation (1)) between two metrics.

Now let us consider a triangle relationship among three measurements such as $\{I_1, I_2, I_4\}$. Assume that we have $I_2 = f(I_1)$ and $I_4 = g(I_2)$, where f and g are both linear functions as shown in Equation (1). Based on the triangle relationship, theoretically we can conclude that $I_4 = g(I_2) = g(f(I_1))$. According to the linear property of functions f and g, the function $g(f(.))$ should be linear as well, which implies that there should exist an invariant relationship between the measurements I_1 and I_4. However, since we use a threshold to filter out those models with low fitness scores, such a linear relationship may not be robust enough to be considered as an invariant. This explains why there is no direct edge between I_1 and I_4.

While each individual invariant models some local relationship between its associated measurements, the network of invariants could essentially capture many invariant constraints underlying large systems. Therefore rather than using one or several models, here we combine large number of invariants into a network to characterize a large system and further use this network for system management tasks. In this section, we discuss how to propagate a value from one node into its equivalent values at other nodes by following the network. Later we need this mechanism to compare the thresholds from various rules.

Without loss of generalization, let us assume that $I_1 = x$. According to Figure 1, we can reach the nodes $\{I_2, I_3\}$ with one hop from I_1. Given $I_1 = x$, the question is how to follow the invariants to estimate other measurements. Since we use the model shown in Equation (1) to extract invariants among measurements, all invariants are the instances of this model template. In Equation (1), if we set the inputs $x(t) = x$ at all time steps, the output $y(t)$ converges to a constant value $y(t) = y$, which can be derived from the following equations:

$$y + a_1 y + \cdots + a_n y = b_0 x + \cdots + b_{m-1} x + b,$$
$$y = \frac{\sum_{i=0}^{m-1} b_i x + b}{1 + \sum_{j=1}^{n} a_j}. \quad (3)$$

Note that the order of invariant models is very small with $n, m \leq 2$ because measurements of computer systems usually have very small temporal delay. With Equation (3), given $I_1 = x$, we can derive the values of I_2 and I_3. Since these measurements are the inputs of other invariants, we can further propagate their values with one hop to I_4 and I_5, and with two hops to I_6. We can not estimate the values of I_7 and I_8 because they are not reachable from I_1. Therefore, given a value at one node, we can follow the invariant network to propagate the value to all other reachable nodes.

5. RULES AND FAULT MODELS

As discussed in Section 1, operators deploy monitoring agents and collect real-time monitoring data to track the operational status of their systems. Rule-based systems are widely used to scan data and trigger alerts for problem determination. However, it is difficult to set good rules and thresholds in complex systems. In practice, operators often set up rules and thresholds based on their system management experiences and intuitions. As a result, alerts originating from various rules often include large number of false positives. For example, upon understanding a rule-based mobile network Operation Support System(OSS) in Asia, we noticed that 80% of all alerts from field reports turn out to be false positives. For 24*365 operational systems like online services, even a very small false positive rate could lead to large number of false positive reports along time. For example, if a measurement (e.g. CPU usage) is checked once per minute and we have 0.1% false positive rate, annually we can still get 24*60*365*0.1% = 526 false positive reports from this single metric. With thousands of such metrics in large systems, it is impossible for operators to analyze all alerts. Large number of false positives often frustrate field operators so that they either ignore the alerts at all or intuitively increase their thresholds to filter out some alerts, which may conversely lead to high false negative rates.

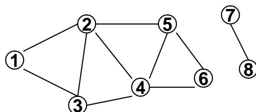

Figure 1: An example of invariant networks

Large systems are often managed by hundreds of operators who have responsibilities for different system segments. Each operator may have his own preference and knowledge to set up rules and thresholds in his local portion of systems. In fact, some system components such as databases may also include management rules provided by their vendors. Since all these rules and thresholds are set in their local context for large number of heterogeneous components, it is difficult to normalize thresholds and manage rules in large computer systems. Due to system dependencies, a single problem may trigger a storm of alerts in large systems. As discussed earlier, many of them might be false positives because of biased threshold setting. Now since we can not compare those rules directly under heterogeneous settings, it is difficult to decide which alerts are important. For example, it is not clear how to compare an alert about CPU usage with another alert on network usage. In a small system, operators may use domain knowledge to decide which category of alerts are more important. For example, an alert from a DNS server might be more important than that from a printer. However, for large systems with huge complexity, such an approach is obviously neither scalable nor practical.

Operators usually have to fix any problems promptly so as to maintain high system reliability and availability. Therefore, it is too time-consuming to analyze every alert without any guidelines and operator may also waste much time on analyzing false positives. To this end, we propose a new approach to rank the importance of alerts, which operators can consult to prioritize their follow-up examinations.

A typical rule consists of a predicate and an action. For example, given a measurement x (e.g. CPU usage), we have such a rule as:

$$\textbf{if}(x > x^T), \textbf{then } generate_alert1,$$

where x^T is the threshold. The predicate can include other single logic condition like "$x < x^T$" or several joint conditions such as "$x > x^T$ && $y > y^T$". The alert may also include text messages to explain itself. Such kind of rule definitions are very common in alarm management of commercial systems [14] [19]. For example, in VMware's alarm management [19], users can define rules to generate alerts with two triggering options: "Is Above (>)" and "Is Below (<)". Default monitor for virtual machine heartbeat is set to

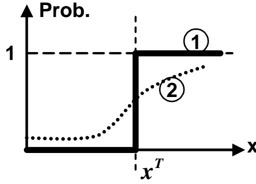

Figure 2: Examples of fault models

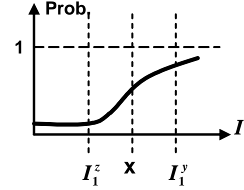

Figure 3: Comparing with multiple thresholds

"Is Below" while default monitor for all other metrics is set to "Is Above". For simplicity, we use several rules as shown in the above example to illustrate our concept. In Section 7, we will incorporate the rules with more complicated predicates into our approach.

Theoretically there is always a fault model behind each rule. Figure 2 illustrates two examples of such conceptual models which are represented by two curves respectively. The x-axis is the value of the measurement x while the y-axis represents the probability of fault occurrence. Since the rule is set with the predicate $x > x^T$, conceptually it implies that the probability of fault occurrence increases (or at least stays constant) with the growth of x. Otherwise operators would have not set such a predicate to generate alerts. Conversely if the predicate is $x < x^T$, the probability of fault occurrence is conceptually expected to increase with the decrease of x. Operators may also set a predicate like $x_1^T < x < x_2^T$ if a fault only occurs during a specific range of x. We will discuss such a joint condition in Section 7 because $x_1^T < x < x_2^T$ can be replaced with two basic logics $x > x_1^T$ and $x < x_2^T$. In this section, we only discuss the fault models underlying the basic predicates like $x > x^T$. The fault model shown in curve 1 (the thick line) represents the ideal situation behind the rule setting, where the probability of fault occurrence is equal to one after a critical value. If this value is chosen as the threshold, we will get no false positives and negatives. However, a more realistic model is shown in curve 2 (the dotted line), where a given threshold will always lead to false positives and/or negatives in problem reporting. Since these theoretic fault models of various measurements are essentially unknown in practice, operators have to select the thresholds based on their experiences and domain knowledge. For example, an operator may use the statistics of historical data to decide a threshold.

6. RANKING ALERTS

In this paper, we do not modify the rules and their thresholds in existing rule-based systems, i.e., the mechanism of generating alerts is not changed at all. In fact, for heterogeneous components in large systems, only the operators administrating them may have the right system knowledge to set up rules and thresholds. For example, we need specific expertise to manage databases or networks. Instead after receiving alerts from various system components, we rank the importance of alerts with a peer review mechanism so that the top ranked alerts are the more trustworthy evidences in problem determination, i.e., we integrate the knowledge from multiple operators to determine the importance of alerts. Precisely, the "importance" here is defined as the Probability of Reporting a True Positive (PRTP). In the following context, we use $Prob(true|x)$ to denote the probability of reporting a true positive under the measurement value x.

Without loss of generalization, lets assume that we have the following set of rules associated with the measurements shown in Figure 1:

1. **if**$(I_1 > x)$, **then** generate $alert1$;
2. **if**$(I_2 > y)$, **then** generate $alert2$;
3. **if**$(I_6 > z)$, **then** generate $alert3$;

where x,y,z are the thresholds. Now given $I_1 = x$, we follow the invariant network shown in Figure 1 to propagate this value and calculate its equivalent values I_2^x at node 2 and I_6^x at node 6 respectively. In the same way, given $I_2 = y$, we can also calculate its equivalent values I_1^y at node 1 and I_6^y at node 6 respectively. In addition, I_1^z and I_2^z can also be calculated in the same way. Now for every measurement listed in the above rules, it has three threshold values including its local threshold and two equivalent threshold values mapped from the other two rules. For example, for the measurement I_1, it has its local threshold x and two equivalent threshold values, I_1^y and I_1^z.

Since the original thresholds x, y and z have different semantics, we can not compare them directly. For example, assuming that x is about CPU usage and y is about network usage, it is meaningless to compare them in different contexts. Now since the other thresholds y and z are mapped into the local context of x, we can compare their equivalent values with x to rank the importance of alerts. Figure 3 illustrates the concept of comparing a measurement with multiple thresholds. Assuming that we have $I_1^z < x < I_1^y$ as shown in the figure, according to the fault model discussed in Section 5, we can conclude that:

$$Prob(true|I_1^z) \le Prob(true|x) \le Prob(true|I_1^y). \quad (4)$$

As discussed in Section 5, since the predicate logic of the above rules is ">", the PRTP will not decrease with the growth of the measurement. Based on this property, we can rank the order of the PRTPs at different thresholds without knowing their real values. Therefore, we do not need a known fault model to rank the PRTPs. Instead, we just need the predicate logics and then compare the equivalent thresholds of the rules to derive their PRTPs' ranking. If the predicate logic of the rules is "<", the order of PRTPs in Inequality (4) should be reversed.

In Figure 3, we rank the thresholds of the rules and their PRTPs in the context of I_1. The question is whether such an order will change in the context of another measurement. For example, what is the order of y, I_2^x and I_2^z in the context of I_2? Figure 4 illustrates these thresholds in different

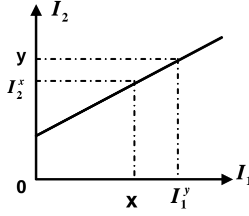

Figure 4: Thresholds in different contexts

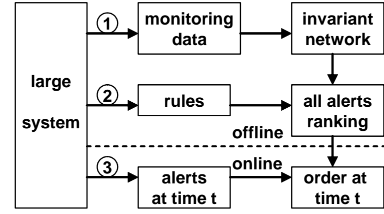

Figure 5: The process of ranking alerts

contexts of measurements. Since I_1 and I_2 have a linear relationship, if $x < I_1^y$ along the axis of I_1, we will also have $y > I_2^x$ along the axis of I_2. Therefore, the order of thresholds will not change even if we map them into different context of measurements. As a result, the order of their PRTPs will not change either. Since we have different fault models for various measurements, the real values of PRTPs calculated in different measurement contexts might be different. For example, $Prob_{I_1}(true|x)$ (calculated with the fault model of I_1) may be different from $Prob_{I_2}(true|I_2^x)$ (calculated with the fault model of I_2) though their thresholds x and I_2^x are equivalent. However, the order of PRTPs will not change no matter which fault models are used here, i.e., if $Prob_{I_1}(true|x) < Prob_{I_1}(true|I_1^y)$ in the context of I_1, we will also have $Prob_{I_2}(true|I_2^x) < Prob_{I_2}(true|y)$ in the context of I_2.

For some large systems, it might be resource-consuming to feed large volume of real-time monitoring data to a central point (e.g. network operation center) for data analysis. We consider two different cases here: Case I, monitoring data is processed by local rule-based agents which forward their alerts rather than the data to the central point, i.e., we only see the alerts but not the monitoring data at the central point; Case II, both monitoring data and alerts are forwarded to the central point. In Case I, we collect historical monitoring data offline to extract invariants and then collect the rules from various system components. Following the extracted invariant network, we calculate the equivalent thresholds of rules and further rank them to decide the importance order of their alerts. For example, with Inequality 4, we rank the alerts with the following descent order of importance: $alert2$, $alert1$ and $alert3$. Since the lowest ranked alerts are likely to be false positives, operators are recommended to double check whether they have set the right thresholds for this portion of alerts.

As discussed earlier, since the order of alerts will not change in different context of measurements, we can map all thresholds into the same context of a single measurement for comparison. Given n thresholds in an invariant network, here we just need $n-1$ mappings to compare these thresholds. Essentially all these steps can be done offline. Now at any time t, after we receive a subset of alerts, we follow the order pre-computed offline to rank this specific subset of alerts. For example, if both $alert2$ and $alert3$ are received at time t, we know that $alert2$ is more important than $alert3$ based on the ranking of all alerts. Figure 5 illustrates this whole process and highlights the offline and online portions of Case I. Note that since real-time data is not available at the central point, we essentially use the static thresholds rather than the current measurement values to rank the importance of alerts, i.e., mathematically we assume that $Prob(true|I, I > x) = Prob(true|x)$, where I is the measurement value and x is its threshold. This follows the fault model represented by curve 1 in Figure 2.

In the above case, we do not consider how much the measurement I deviates from its threshold x. In general, the curve 2 shown in Figure 2 illustrates that $Prob(true|I, I > x)$ increases as the value of I grows. If both measurement values and alerts are available at the central point, we should use the real values rather than their thresholds to rank alerts. In this case (denoted by Case II), as shown in Figure 3, we compare the measurement value against all equivalent thresholds to determine the Number of Threshold Violations (NTV), i.e., we map the thresholds of all other rules into the local context of a measurement and then check whether its value violates each mapped threshold. Note that we only calculate the NTV of a measurement after it generates an alert, i.e., this measurement at least violates its own local threshold so that its NTV is at least 1. For example, in Figure 3, if $x < I_1 < I_1^y$, its NTV is 2 because I_1 is larger than the thresholds x and I_1^z. If $I_1 > I_1^y$, its NTV is 3. In the same way, we can calculate the NTVs of other measurements I_2 and I_6. Since the order of equivalent thresholds remain the same in different context of measurements, they are essentially used as the reference positions for us to compare the measurement values in different contexts and later we use their NTVs to rank the alerts. If a measurement value violates more equivalent thresholds, its alert has a higher PRTP and it is more important in the follow-up examinations. Besides its local rule, the NTV essentially represents the number of other rules (peers) that agree with such an alert.

In Case II, we follow the same offline steps shown in Figure 5 to collect monitoring data and rules, extract invariants and calculate the equivalent threshold values. However, given n measurements with thresholds in an invariant network, each threshold is mapped into the context of the other $n-1$ measurements and we have total $n(n-1)$ mappings. As a result, each measurement has n thresholds including its local threshold and $n-1$ equivalent thresholds mapped from other rules. Note that the computational overhead of such a mapping as shown in Equation (3) is negligible and all these steps are also done offline. Each measurement has a vector to store the equivalent thresholds in its context. Now at time t, after we receive a set of alerts online, the measurement value associated with each alert is compared with all equivalent thresholds in its local context to determine its NTV. We then sort their NTVs to rank the importance of alerts. Figure 6 summarizes the major steps of online alert ranking method used in Case II.

The difference between Case I and Case II is whether we

Algorithm 6.1: Online alert ranking

1. Collect historical monitoring data from systems;
2. Extract invariants as illustrated in Section 3;
3. Collect management rules from systems;
4. Compute all equivalent thresholds as shown in Section 4;
5. At time t, receive a set of alerts from systems;
6. Compare the measurement value of each alert with its vector of equivalent thresholds to calculate NTV;
7. Sort NTVs to rank the set of received alerts;
8. Go back to Step 5.

Figure 6: Online alert ranking method

use measurement values to rank alerts. In Case II, the central point receives the alerts as well as real measurement data. Conversely in Case I, only the alerts are forwarded to the central point. However, our approaches in two cases can be unified with Algorithm 6.1. At step 6, if we replace the measurement value with its local threshold value, essentially we can compare this local threshold with the equivalent thresholds of other rules to get the NTV. Now if we sort the NTVs, we will get the same order of alerts as that resulting from Case I. This is because the equivalent thresholds are used as the reference positions in comparison and they have the same order even in different context of measurements. Some rules may have dynamic thresholding mechanism. In this case, for every new updated threshold, the step 4 of Algorithm 6.1 should re-propagate its new value to the other $n-1$ nodes.

In Case II, it seems that we can map the measurement values into the same context and compare them directly to rank alerts. However, when many rules are violated under various system faults, their measurements may not follow the original invariant relationships anymore. Therefore, we can not use the same invariant network to propagate a value from one node to others at this time. In fact, if we still have the same invariant network, the real values observed at two nodes should exactly reflect their mapping relationship and they are already "equivalent", i.e., we can just observe the real values rather than map their values. Instead, in Algorithm 6.1, it is the threshold value that propagates through the invariant network because the thresholds of various rules are arbitrarily set and do not follow those invariant relationships. Conversely measurement values are observed from real systems and they naturally follow the physical constraints of their underlying systems.

Real measurements are locally compared to the equivalent thresholds that are mapped from other rules. Note that these equivalent thresholds are mathematically derived but do not exist in real systems. They are essentially used as the references to compare the aggressiveness or conservativeness of operators' threshold selection practice. Even if some invariants will not hold at the threshold values in reality, the "virtual" equivalent thresholds can still be calculated in the same way and used as the references to compare the "tightness" of various rules. In addition, it is unnecessary to rank alerts with small difference. For example, if the measurement values of two alerts both violate all equivalent thresholds, they are equally critical though their real values may have some differences. Therefore, in Algorithm 6.1, we use the NTVs rather than the measurement values to rank alerts.

7. EXTENSIONS AND LIMITATIONS

In current rule-based systems, each measurement is compared with its own threshold to generate alerts and each rule works in its isolated local context. In the above section, such a measurement is further compared with the equivalent thresholds mapped from other rules to determine the importance of its alerts. Therefore, with invariant networks, we are enabled to bring individual thresholds into a global context. Essentially we introduce a peer-review mechanism so that a measurement is not only checked by its own rule but also other rules from its peers. Alerts are ranked based on the NTVs, which represent how many peers agree with a local threshold. Obviously an alert with higher NTVs should be ranked more important because more peers would have generated such an alert by themselves. While each threshold might not be precisely set by individual operators, our approach can boost problem reporting accuracy by introducing such a collaborative peer-review mechanism to integrate the knowledge from many operators. Our motivation is that the consensus from a group of rules is more accurate than an individual one, whose threshold might be much biased. Though we use the "fault models" in Section 5 to illustrate our concept, our approach can also be used to manage alerts for other tasks such as performance and security management by replacing fault models with attack or anomaly models.

In the previous sections, we assume that the predicate logic of all rules is "$x > x^T$" (denoted by "$>$" logic), where x is the measurement and x^T is its threshold. As discussed earlier, if the predicate logic of rules is "$x < x^T$" (denoted by "$<$" logic), our approach and Algorithm 6.1 remain the same. However, under "$<$" logic, if a measurement value gets smaller, its NTVs will be higher, which is reversed under "$>$" logic. In practice, some rules have "$>$" logic while others may have "$<$" logic. The question is how to rank alerts from the mixture of these rules. As discussed in Section 6, since the fault models behind various measurements are unknown, we rank the PRTPs of alerts without knowing their real values. However, due to different fault models, we can not rank the PRTPs by comparing one threshold in "$>$" logic with another one in "$<$" logic, i.e., the order of PRTPs can not be directly derived from the order of thresholds. For example, under "$>$" logic, the PRTP increases as the measurement value grows. Conversely, under "$<$" logic, the PRTP decreases as the measurement value grows. We can only rank these alerts with their real PRTP values which are unknown in practice. Therefore, based on the predicate logics of rules, we have to split alerts into two clusters and rank them separately. In fact, rules with "$>$" or "$<$" are set to monitor different system states such as system overloading or system down. At a specific system state, many of alerts may only belong to one single cluster. As shown in VMware's example in Section 5, all metrics except VM heartbeat have rules set with "$>$" logic to track performance problems while VM heartbeat is

used to check the liveness of a VM.

The predicate of some rules may also include joint conditions such as "$x_1^T < x < x_2^T$" and "$x > x^T$ && $y > y^T$". The same question is how to rank the alerts from such rules with others. A joint condition can always be composed with several basic ">" and "<" logics. For example, "$x_1^T < x < x_2^T$" can be rewritten as "$x > x_1^T$" and "$x < x_2^T$". For all ">" logics in a joint "and" condition, the logic with the highest equivalent threshold is used to rank its alert because it subsumes all other ">" logics. Conversely, for all "<" logics in a joint "and" condition, the logic with the lowest equivalent threshold is used to rank its alert because it subsumes all other "<" logics. For multiple logics in a joint "or" condition, we can use the measurement values to determine which logics are satisfied and only these logics are then used to rank the alert associated with this condition. Note that a joint "or" condition is not common in practice because it can not distinguish different scenarios in problem reporting. Therefore, we can always convert multiple ">" logics (or "<" logics) in a joint condition into a single ">" logic (or "<" logic). If a joint condition includes both ">" and "<" logics, we rank the single alert among both clusters of ">" and "<" alerts. In conclusion, a rule with a joint condition can be automatically converted into a rule with a single ">" logic and/or a single "<" logic, which can be further ranked with other rules.

As shown in Figure 1, there may exist several disconnected invariant networks and not every measurement node is always reachable from another one. One limitation of our approach is that we can not rank the alerts originating from the measurements that are not reachable from each other. As discussed earlier, it is the invariant network that essentially enables us to estimate equivalent threshold values and further rank the importance of alerts. Therefore, we can only rank the alerts from the measurements within the same subnetwork but not across disconnected subnetworks. However, as discussed in Section 3, in practice most of measurements belong to the same invariant network because they respond to the volume of external workload accordingly. Meantime, compared to analyze each alert separately, it is still meaningful to rank alerts within each invariant subnetworks and further analyze every cluster of alerts with their importance order.

8. EXPERIMENTS

In this section, we report results from several experiments that verify the feasibility of our approach. Since there are no standards and benchmarks of target systems, monitoring data, rules and faults, it is difficult to evaluate the accuracy of our approach. For example, different set of monitoring metrics and faults may lead to different accuracy in problem reporting. In the future, we plan to run our solution over commercial systems for a long period (e.g. 6 months) so as to evaluate its accuracy based on operators' field reports. Otherwise it is really hard to simulate large number of real problems encountered in system management practice. Our experiments were performed in a typical three-tier web system which includes an Apache web server, a JBoss application server [8] and a MySQL database server. Figure 7 illustrates the architecture of our experimental system and its components. The application software running on this system is Pet store [18], developed by Sun Microsystems.

In our previous work [9], we collected as many as 111

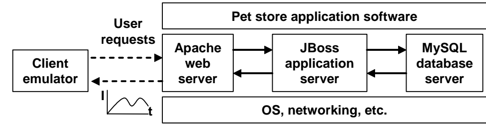

Figure 7: The experimental system

measurements from three servers and extracted 975 invariants from this test bed system. However it is difficult to see the connectivities in the meshed invariant network. In this paper, we just use 10 measurements to extract a small invariant network so that we can have a microscopic view on their relationships. For each server, we monitor its CPU usage as well as its numbers of network packets that are received and sent respectively during every 6 second sampling unit. We chose these metrics because it is easy to monitor them and collect data. In addition, we also monitor the number of HTTP requests per sampling unit because it measures the volume of user workloads. For 10 measurements, totally we extract 41 invariants after taking fitness score threshold $\widetilde{F} = 0.6$. Figure 8 illustrates the invariant network extracted from these 10 measurements whose initial tag "Web", "AP" and "DB" represent the web server, the application server and the database server respectively.

Six measurements are chosen to set rules with the ">" logic. They are "Web_CPU%","AP_CPU%","DB_CPU%", "Web_Recv_Packet", "AP_Recv_Packet", "DB_Recv_Packet", whose threshold values are set to be 70%, 30000, 80%, 30000, 70% and 20000 respectively. The application server runs bulk of business logic so that we set its CPU threshold higher than that of the other two servers. For convenience, we denote the above six measurements by $m_1, m_2, \cdots, m_6$ and their thresholds by $m_1^T, m_2^T, \cdots, m_6^T$ respectively. In addition, we denote the alerts associated with $m_i (1 \leq i \leq 6)$ by A_i respectively. Table 1 illustrates the thresholds of these six measurements and their equivalent thresholds calculated in the context of other measurements. Note that the equivalent thresholds are calculated using the invariant network shown in Figure 8. In this table, the equivalent threshold value at the ith row and the jth column $(2 \leq i, j \leq 7)$ results from mapping the local threshold m_i^T into the context of m_j. The diagonal elements of the table are the local thresholds of m_i (the numbers in bold fonts). For example, the local threshold of m_3 is 80 and its equivalent threshold in the context of m_1 is 70.2.

As discussed earlier, since various measurements have dif-

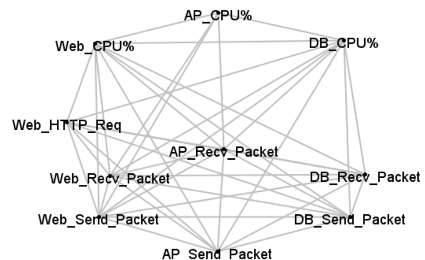

Figure 8: Extracted invariant network

Table 1: Thresholds of measurements

	m_1	m_2	m_3	m_4	m_5	m_6
m_1^T	**70**	32726	78	29540	62.8	23208
m_2^T	63.6	**30000**	71.4	27018	57.4	21200
m_3^T	70.2	33006	**80**	29646	63	23291
m_4^T	70.5	33212	81	**30000**	63.7	23509
m_5^T	77	36316	86.4	32613	**70**	25688
m_6^T	59.8	28207	66.9	25469	54.1	**20000**

ferent semantics, we can not compare their thresholds directly. For example, it is meaningless to compare $m_1^T = 70\%$ (CPU usage) with $m_2^T = 30000$ (packet number). With the invariant network shown in Figure 8, we can map the local thresholds into the context of other measurements. In Table 1, all equivalent thresholds within the same columns have the same context and can be sorted to rank the importance of alerts. As a result, we have the same descent order of equivalent thresholds from each column: m_5^T, m_4^T, m_3^T, m_1^T, m_2^T, m_6^T. This verifies that the ranking order of equivalent thresholds does not change in different context of measurements, as discussed in Section 6. Now if we only receive alerts but not the measurement values at the central point (i.e., Case I in Section 6), since the logic of our rules is ">", the importance of alerts should be ranked with the following descent order: A_5, A_4, A_3, A_1, A_2, A_6. All the previous steps can be done offline. Now during online operation, we follow this order to rank the subset of alerts received at each time step t.

Now we use the alerts received at two time points to illustrate how to rank alerts with their measurement values. Table 2 includes the measurement values and their NTVs at two time points marked with $t2$ and $t3$ respectively. In the last column, "-" means that there is no alert from that measurement, i.e., the measurement value is below its own local threshold. As described in Algorithm 6.1, NTVs are calculated by comparing each measurement value with its vector of equivalent thresholds, which are the columns in Table 2. For example, at time $t2$, m_1's real value 73.6 is compared to the second column (i.e. m_1 column) to calculate its NTV.

At time $t2$, we receive alerts from every rule. Based on each measurement's NTV, we rank the alerts with following descent order: $A_5, \{A_1, A_2, A_3, A_4\}, A_6$. At this time point, in fact we observe slow database response time from our system though we do not know its root cause. Note that we observe this anomaly from our test bed system without injecting any problems. Figure 9 shows the real measurement curves of "DB_CPU%"(m_5) and "AP_CPU%"(m_3). In this figure, y-axis is the average percentage of CPU usage while x-axis represents the time points with 6 second sampling interval. Since we want to analyze these curves with great details, we have only plotted the curves of these two measurements here.

Table 2: Ranking alerts with NTVs

time	$t2$		$t3$	
metrics	value	NTVs	value	NTVs
m_1	73.6	5	73.5	5
m_2	34319	5	31478	2
m_3	81.6	5	54.6	-
m_4	30621	5	22712	-
m_5	71.4	6	46.1	-
m_6	22620	2	18564	-

As shown in Figure 9, between the time points $t1$ and $t2$, we only see the alert A_3 from the measurement "AP_CPU%". At the time point $t2$, we observe the alerts A_3 and A_5 from both measurements. According to Table 2, A_5 is ranked more important than A_3 at this time point though both of them have high NTVs. In order to verify that, we follow our previous model-based detection method proposed in [10] to plot the expected values of these measurements, which are shown as the solid lines in Figure 9. The dotted lines represent the real values observed from our system. Given the workloads measured from our system (i.e., Web_HTTP_Req in Figure 8), we follow the invariant relationships to calculate the expected values of "DB_CPU%" and "AP_CPU%". For example, with an invariant $y = f(x)$, given the time series of x, we can follow the equation to calculate the expected time series of y, which might be different from its observed measurement values. In our model-based fault detection approach, we report an anomaly if the difference between the expected value and observed value of a measurement is large. In Figure 9, the dotted line of "DB_CPU%" matches its solid line very well until we see a big deviation of two lines at $t2$. Conversely, many deviation points are observed for the two lines of "AP_CPU%" because the application server runs intensive application logic and has much dynamics. However, around the time point $t2$, we do not observe much deviation of two lines. Therefore, our observation of slow database response and the sudden big deviation of "DB_CPU%" two lines at time $t2$ both confirm that the alert A_5 is more likely reporting a true anomaly, which matches our ranking very well.

At time $t2$, we observe the real anomaly from our system without knowing its ground truth. At time $t3$, we simulate a problem by suddenly copying a big file from a user machine to the web server with SCP(secure copy) command. Since we know the root cause of the problem in this case, we can further verify the effectiveness of our approach. At time $t3$, we only receive the alerts from "Web_CPU%" and "Web_Recv_Packet". The real values of six measurements

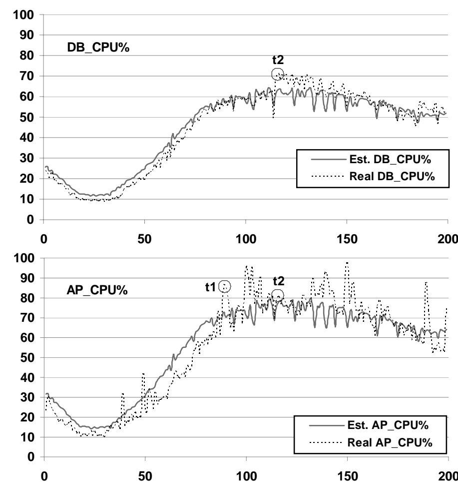

Figure 9: Measurement values at time t2

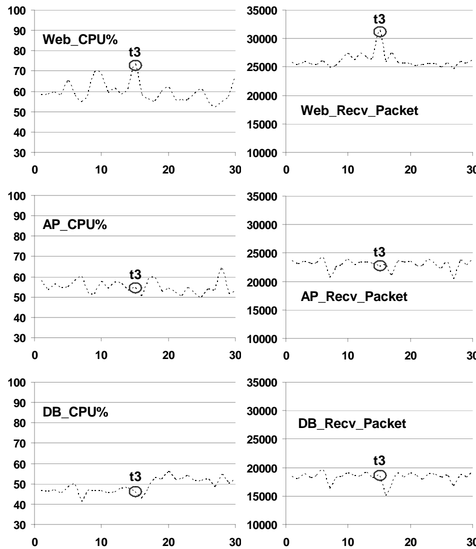

Figure 10: Measurement values at time t3

are shown in Table 2 and only the values of m_1 and m_2 are higher than their local thresholds. We do not see alerts from the other four measurements so that it is not necessary to calculate their NTVs. Therefore, based on their NTV ranking in Table 2, we rank the alerts at time $t3$ with A_1, A_2. Figure 10 shows the curves of six measurements and we only observe anomalies at time $t3$ from the curves of "Web_CPU%" and "Web_Recv_Packet". The y-axis of the curves at the right column of this figure represents the number of network packets. It is easy to understand that both measurements are expected to be affected in this case.

Note that we can not directly link the importance of alerts with root-cause localization. As shown in the above two cases, we can report the abnormal metrics but it still needs operator's analysis to locate the root-cause. For example, a root cause might be a line of buggy code in a software. However, like a crime scene investigation, operators need to follow these abnormal metrics to narrow down the faulty segments and further locate the root cause with fine-granularity tools. As discussed earlier, the importance of alerts is ranked by their probability of reporting a true anomaly. The top ranked alerts are more likely to be true positives. Operators should use the top ranked alerts as the trustworthy evidences to localize the problem.

9. CONCLUSIONS

Rule-based systems are widely deployed in practice for operational system management. In this paper, we use invariant networks to map the local thresholds of various rules into their equivalent values in a global context to rank alerts. Essentially we introduce a peer review mechanism so that a measurement is not only compared to its local threshold but also the equivalent thresholds mapped from other rules to determine the importance of its alerts. The top ranked alerts are more important because they get more consensus from other rules. Operators can consult the order of alerts to prioritize their problem determination process.

10. REFERENCES

[1] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen. Performance debugging for distributed systems of black boxes. In *Proceedings of ACM SOSP*, pages 74–89, NY, 2003.

[2] blog.wired.com/business/2008/02/customers-shrug.html.

[3] money.cnn.com/2007/04/18/technology/rimm/.

[4] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer. Path-based failure and evolution management. In *Proceedings of USENIX NSDI*, pages 309–322, San Francisco, CA, 2004.

[5] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox. Capturing, indexing, clustering, and retrieving system history. *SIGOPS Oper. Syst. Rev.*, 39(5):105–118, 2005.

[6] B. Gruschke. Integrated event management: event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE DSOM*, Newark, NJ, 1998.

[7] Z. Guo, G. Jiang, H. Chen, and K. Yoshihira. Tracking probabilistic correlation of monitoring data for fault detection in complex systems. In *Proceedings of 2006 DSN*, pages 259–268, 2006.

[8] www.jboss.com/.

[9] G. Jiang, H. Chen, and K. Yoshihira. Discovering likely invariants of distributed transaction systems for autonomic system management. In *Proceedings of the 3rd ICAC*, pages 199–208, Dublin, Ireland, 2006.

[10] G. Jiang, H. Chen, and K. Yoshihira. Modeling and tracking of transaction flow dynamics for fault detection in complex systems. *IEEE Trans. Dependable Sec. Comput.*, 3(4):312–326, 2006.

[11] G. Jiang, H. Chen, and K. Yoshihira. Efficient and scalable algorithms for inferring likely invariants in distributed systems. *IEEE Trans. on Knowledge and Data Engineering*, 19(11):1508–1523, 2007.

[12] L. Ljung. *System Identification - Theory for The User.* Prentice Hall PTR, second edition, 1998.

[13] C. Mas and J.-Y. L. Boudec. An alarm filtering algorithm for optical communication networks. In *Chapter 18, Management of Multimedia Networks and Services.* Springer, 1998.

[14] www.novell.com/coolsolutions/appnote/16897.html.

[15] D. Oppenheimer, A. Ganapathi, and D. Patterson. Why do internet services fail, and what can be done about it. In *Proceedings of of the 4th USITS*, pages 1–16, Seattle, WA, 2003.

[16] J. Parekh, G. Jung, G. Swint, C. Pu, and A. Sahai. Comparison of performance analysis approaches for bottleneck detection in multi-tier enterprise applications. In *Proceedings of IEEE IWQoS*, pages 302–306, CT, 2006.

[17] D. Patterson. A simple way to estimate the cost of downtime. In *Proceedings of LISA-2002*, pages 185–188, Philadelphia, PA, 2002.

[18] java.sun.com/developer/releases/petstore/.

[19] www.vmware.com/support/vc11/doc/c15alarms.html.

[20] A. Yemini and S. Kliger. High speed and robust event correlation. *IEEE Communication Magazine*, 34(5):82–90, May 1996.