Using Time over Threshold to Reduce Noise in Performance and Fault Management Systems

Mark Sylor[1] and Lingmin Meng[2]

[1]Concord Communications, 600 Nickerson Rd. Marlboro, MA 01752
sylor@concord.com

[2]Department of Electrical and Computer Engineering, 8 St. Mary Street
Boston University, Boston, MA 02215

Abstract: Fault management systems detect performance problems and intermittent failures by periodically examining a metric (such as the utilization of a link), and raising an alarm if the value is above a threshold. Such systems can generate numerous alarms. Various schemes have been proposed for reducing the number of alarms, or filtering out the important ones. The time over threshold detection algorithm reduces the volume of alarms at the source detector. This paper describes an experiment that compares time over threshold against simple threshold crossings. The experiment demonstrates that it reduces the number of alarms raised by a factor of 25 to 1 without any significant reduction in the problems detected.

1 Introduction

Concord s Network Health“ product, a member of the eHealth“ product suite, collects performance and fault management information from networks, systems, and applications. It stores the information for historical analysis, and presents the information in report format on the web or on paper. The information collected is used to analyze the overall health of the networks, systems, and applications, and to support capacity planning.

Concord has recently added LiveHealth to the eHealth product family. LiveHealth analyzes the information Network Health already collects in real time to detect faults and performance problems. Because Network Health maintains a historical record of past performance and faults, it was natural to use that data to improve the detection capabilities of LiveHealth.

Most fault and performance management systems depend on simple threshold crossing events to detect problems. They periodically sample the value of some performance or fault metric, and compare it with a fixed value, called a threshold. If the value of the metric is greater (or less) than the threshold, an alarm is raised. When the alarm is raised, a notification is sent to a network management system (an NMS), generally in the form of an SNMP trap, a CMIP notification, or in some proprietary format.

While simple threshold crossings are effective in detecting problems, they generate far too many alarms. The metrics indicative of performance problems show a wide variation, with little predictability in their pattern. For example, Fig. 3, shows a graph of

A. Ambler, S.B.Calo, and G. Kar (Eds.): DSOM 2000, LNCS 1960, pp. 145 - 156, 2000.

the utilization of a link over three days measured at 5 minute intervals. As is well known, such metrics vary widely. Other metrics, discarded packets, error rates, congestion indications, disk I/O rates, CPU Utilization, application workloads (transaction rates), all show similar high variation. This variation guarantees that some samples will be above any achievable threshold. No matter how high the threshold is set, sooner or later, that threshold will be exceeded and an alarm will be raised. These alarms are a form of false alarm.

False alarms have a serious impact on any real time fault or performance management system. If there are too many alarms, operators will tend to ignore them all, including the alarms that indicate real problems. Even if the operator is conscientious, finding a particular alarm from a list of thousands of alarms is difficult.

One approach to dealing with this flood of alarms is to filter, classify, and prioritize the alarms in the network management system receiving the notifications. For example, an NMS might filter out unimportant alarms based on a severity field included in the notification. An NMS might classify the alarms based on fields in the notification, the element (object, or host) raising the alarm, the type of alarm, the variable which exceeded the threshold, and other fields. Once classified, an NMS might simply count the number of events of a class. Based on the count or the class, the NMS might prioritize the alarm, or take action on the alarm. The kinds of actions an NMS might take include actions to notify an operator (for example by paging the operator). Another is to change the state of the NMS, for example, receiving an alarm of a particular class causes a change in the rules so subsequent alarms of that class are discarded. While all these are appropriate actions, they can be difficult to set up, and may not reduce the overall alarm rate.

With LiveHealth, we attempt to decrease the flood of alarms at the point of detection, rather than provide a better means of handling a flood of alarms generated by the detector. The technique adopted for reducing alarms is based on a heuristic detection algorithm called *time over threshold*.

2 Related Work

A common technique used to decrease the flood of alarms has been to use thresholds to drive an alarm state. When the threshold is exceeded, an alarm is raised. When the value falls below a threshold, the alarm is cleared. In some of these systems, the falling threshold can have a different value than the rising threshold. This technique was standardized in [5], implemented in commercial Network Management Systems such as [3], and implemented in agents within network devices such as [6]. While these techniques help, experience shows they do not reduce the alarms enough. Further, they depend on setting the falling threshold correctly, yet there is no obvious value that is appropriate.

More recent work has focused on using statistical approaches to setting the threshold [7], or in using statistical techniques to detect points of change that may indicate a fault [2]. These approaches improve the quality of alarms by more accurately predicting when an alarm is likely. Part of the implementation of Live Health exploits very similar techniques to set better thresholds. This work is not covered in this paper. Our

experience so far is that these approaches do not lower the alarm rate, in fact they tend to increase the number of alarms, as they detect problems that previously were missed. Approaches combining improvements in detecting problems (such as statistical thresholds) with good noise reduction techniques such as described here are needed to provide users with high quality alarms.

3 Operation of Network Health

Network Health periodically polls counters from elements (managed objects) in the network. It computes and stores the differences in the counters between the samples, and the difference in time between the samples, which is stored in the database as a *statistics record*. From the statistics record, performance and fault metrics (called trend variables in Network Health) can be computed. These trend variables are often rates, such as *bytes sent per second*. The value of a rate trend variable is the average rate over the interval covered by the sample period. From these basic statistics records, and trend variables, Network Health computes numerous reports on the performance of the network, systems, and applications.

With the addition of Live Health, Network Health takes those same statistics records, and passes them to an evaluation engine called the LiveExceptions Engine (LE engine) in addition to, or instead of, the database. The purpose of the LE engine is to detect performance or fault problems. The LE engine detects problems by evaluating current metrics against a set of rules. Live Health includes a rich collection of default rules. When a rule detects a problem, the LE engine raises an alarm. When the rule detects that the problem has gone away, the alarm is cleared. Alarms are displayed in the Live Exceptions alarm browser. An alarm can be sent as a trap to an NMS where it is displayed with other events, or can be used to drive the status (color) of objects in the NMS map.

4 Time over Threshold

The time over threshold algorithm is implemented in the LiveExceptions Engine. When a statistics sample is received for an element the engine analyses the statistics against all the applicable rules. Each rule defines a detection algorithm to apply to the data, and any parameters used to control the algorithm. The time over threshold algorithm computes the value of a trend variable over a time period called the analysis window, and compares it with a threshold. It then determines how long the variable was over the threshold. If that the variable was over threshold for a length is greater than an alarm window, then an alarm is active for the sample period. When an alarm changes from inactive to active, we say the alarm is raised. When an alarm changes from active to inactive, we say the alarm is cleared.

A typical example is an alarm on the CPU Utilization of a Unix server. We raise an alarm if the CPU Utilization is greater than 90% (the threshold) for more than 15 minutes (the alarm window) out of the past hour (the analysis window).

More formally, the time over threshold algorithm can be defined as follows.

Let R be a time over threshold alarm rule defined for a trend variable $x(t)$, where the rule defines a threshold, T, an analysis window, W, and an alarm window A.

Assume at time t_n the engine receives a new statistics sample x_n of the trend variable $x(t)$ and this sample covers the period $t_{n-1} < t \le t_n$, i.e., for all $i = 1..n$

$$x(t) = x_i \text{ for all } t \text{ such that } t_{i-1} < t \le t_i$$

Further, assume that the samples x_i for $i = j..n$ cover the analysis window, i.e.,

$$t_{j-1} < (t_n - W) \le t_j < t_{j+1} < ... < t_n$$

Let the threshold state of sample i, c_i represent whether x_i exceeds the threshold, c_i be defined as

$$c_i = \begin{cases} 1, & x_i > T \\ 0, & \text{otherwise} \end{cases}$$

Then we compute, b_n, the time $x(t)$ is over the threshold T in the analysis window W for sample n as

$$b_n = (t_j - (t_n - W))c_j + \sum_{i=j+1}^{n} (t_i - t_{i-1})c_i$$

Finally, let a_n represent the alarm state of a rule for sample n be defined as

$$a_n = \begin{cases} 1, & b_n \ge A \\ 0, & \text{otherwise} \end{cases}$$

We use the alarm state to raise and clear alarms as follows. If $a_{n-1} = 0$ and $a_n = 1$ then we raise an alarm at time t_n. If $a_{m-1} = 0$ and $a_m = 1$ then we clear that alarm at time t_m. If $a_n = 0$ we say the alarm is inactive, if $a_n = 1$, we say the alarm is active.

This definition describes the basic idea behind time over threshold. We have generalized the definition in two ways.

First, the time over threshold computation of the condition state can use more general expressions to compute the threshold state. Live Exceptions supports expressions such as

$$(x_i < T)$$
$$((x_i > T) \,\&\, (y_i > S))$$

Live exceptions also supports dynamically computed thresholds based on long term historical analysis of the behavior of the variable. That part of the work is not covered here.

Second, the samples may not perfectly cover the analysis window. Failures in the polling process or in the devices being monitored can lead to gaps in the data. When the system initially starts monitoring a rule, there is some start up period where the samples will only cover a portion of the analysis window. In these cases, the threshold state $a(t)$ of any period that is uncovered by a sample is assumed to be 0 (false).

The time over threshold algorithm is similar to, but not the same as an algorithm based on the number of samples over the threshold. We used time rather than samples for a number of reasons.

- Samples cannot be collected on precisely regular intervals. A small jitter in the time between polls on the order of a few seconds is introduced because:
  - The network introduces delays in packet latency.
  - Other activities on the system running Network Health add jitter to the sampling process.
  - Other activities in the system being monitored generate jitter in the sampling periods.
- Polls may be lost.
  - Communications problems in the network can cause SNMP requests or responses to be lost.
  - Agents may sometimes fail to respond. An agent may have limited memory to buffer requests or have other limitations that cause them to drop SNMP requests.

  Network Health can and does recover the data for those missed polls. When it does recover the data, the resulting sample covers two or more of the scheduled sample periods.

For these reasons, using the time each sample covered, rather than the number of samples collected is a better measure of behavior.

The time over threshold algorithm is designed to handle a number of common behaviors in trend variables.

Many variables experience isolated spikes such as those in Fig. 4 that graphs the percentage of frames which had an error on a WAN link. In many cases, a single, isolated spike is not a real problem. Only when enough samples are bad should an alarm be raised. By setting the alarm window, A, to a longer period (say 15 minutes), we can ensure that an alarm is raised only when the problem persists long enough to impact the system.

When a variable crosses over a threshold, there are likely to be periods when the variable will fall below the threshold for a few samples, only to return above the threshold shortly thereafter. The analysis window (or more precisely, the analysis window, less the alarm window) controls how long the variable must remain below the threshold before the alarm is cleared. In general, increasing the analysis window reduces the probability that when an alarm is cleared, it will simply be raised again within a short time. Fig. 5 shows a typical situation where a variable crosses above a threshold and stays above for most of the time, but occasionally falls below the threshold. In this case the time over threshold raises the alarm at the beginning of the problem, and keeps it active throughout the period.

5 Experiment

As we were developing the set of default rules used to detect problems we ran numerous experiments on many live networks, and on saved databases of data collected by Network Health. We developed a tool to replay a database by reading the collected samples, and passing them to the LE engine as if they had been polled. This tool allowed us to compare the behavior of different rule sets, and to fine-tune the thresholds and parameters that control the rules. The tool also allowed us to evaluate and compare the detection algorithms with more conventional methods against data collected from real networks. One of these experiments is reported here.

The database used in the experiment covers a period slightly more than three days of monitored data. Each element was polled at a 5 minutes rate. The number of samples for each rule is approximately proportional to the time duration, in this case about 900 samples. 1274 elements were monitored, including networks, systems, and applications. The types of elements monitored included: 43 routers or switches, 9 servers, each running from 1 to 3 applications, 5 Network Access Servers, 69 frame relay circuits, 68 LANs, 326 WAN links, and 2 ATM channels. Each element was evaluated against the complete set of default performance and fault rules. Each element had from 5 to 15 alarm rules applied to it. The network is a fairly typical enterprise network. Although only a portion was monitored, the elements were representative of the whole network.

The goal of this experiment was to compare the detection effectiveness of three algorithms: Time over Threshold (TOT), Threshold Driven State (TDS), and Simple Thresholds (ST).

The Simple Thresholds (ST) algorithm simply tests the sampled trend variable against the threshold, and raises an alarm whenever the variable is above the threshold. Because a trap is sent for every sample over threshold, it does not send alarm clear traps. One apparent problem with ST is that it raises so many alarms that the important information to the system manager could be buried in the flood of alarms. The simple threshold forms a basis of comparison that any algorithm can be evaluated against.

The Threshold Driven State (TDS) algorithm attempts to compensate for the problems in ST by remembering the state of the alarm for the previous sample, and sending a trap when the state changes. The TDS algorithm can reduce the number of alarms raised for when an alarm is caused by a consecutive sequence of bad polls, such as seen in Fig. 5. However, it has problem when the value bounces up and down crossing the value

threshold frequently such as that shown in Fig. 6. Variables with high variability around the threshold cause TDS alarms that babble. One fix to TDS that has been proposed is to use a lower threshold to clear the alarm than the threshold used to raise the alarm. It is difficult to find a good clearing threshold. Consider for example the system shown in Fig. 7, here TDS would raise an alarm on each of the spikes over the threshold, and clear it on the next sample. Each alarm would be a false alarm, and no reasonable setting of a falling threshold would correct that problem.

To compare the three algorithms, we ran two replays of the database. First we ran the default rules using the standard TOT windows against the database. Most of these rules use an alarm window, *A*, of 15 minutes, and an analysis window, *W*, of 60 minutes.

For the second run, observe that if the TOT algorithm is run on rules where $A = W$ is less than the minimum sample period, then by the definitions above, the alarm state a_n equals the threshold state c_n of the sample. Since both the ST and TDS algorithms are based on the threshold state, we can reconstruct how the algorithms will behave. In particular, by setting both the analysis window size and the alarm window size to 1, we are able to reconstruct the original binary information whether the variable at each time interval is above or below the threshold. We then processed this binary information to determine the behavior of the ST and TDS algorithms.

6 Results

We compute the following 5 performance parameters for each algorithm to determine their effectiveness at detecting problems.

1. The number of bad samples (samples when the monitored variable is above the threshold) covered by the raised alarms.
2. The number of good samples (samples when the monitored variable is within the normal range) covered by the raised alarms.
3. The number of alarm set traps each algorithm sends.
4. The number of alarm clear traps each algorithm sends.
5. The total number of traps (both alarm set and alarm clear) each algorithm sends. The traps draw the network manager or administrators attention. This number should be as small as possible to reflect only the real network outages or potential problems.

The following table summarizes the performances of the three algorithms, for all the elements and all of the rules.

Table 1. Comparison of Algorithms

	Simple Thresholds (ST)	Threshold Driven State (TDS)	Time Over Threshold (TOT)
# bad polls covered	17505	17505	16373
# good polls	0	0	10131
# alarm set	17505	3836	709
# alarm clear	0	3836	709
# traps (set or clear)	17505	7672	1418

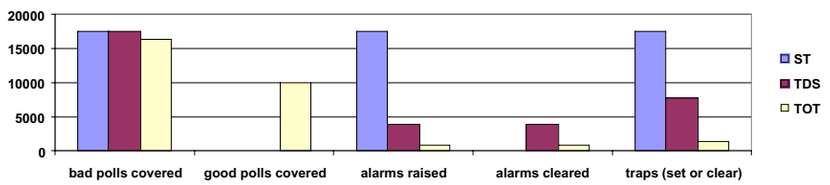
The ST and TDS algorithms raised alarms for 223 elements, while the TOT algorithm raised alarms for 158 elements. An example of a case where TOT did **not** fire an alarm is that shown in Fig. 4. These isolated spikes were a source of many false alarms.

The TOT algorithm covered 93.5% (16373/17505) of all the bad polls, or equivalently, 93.5% of the time when the variable is above the value threshold. The number of alarm sets was reduced by a factor of 25 (17505/709) from ST or 5.4 (3836/709) from TDS.

Note that alarms raised by the TOT algorithm covered a small number of good polls (10131). These polls lie in the gaps between bad polls and serve to reduce the total number of raised alarms. This number also indicates the frequency of bad polls during the alarm periods raised by the TOT algorithm. The frequency of bad polls vs. good polls during these periods is 8:5 (16373:10131).

The number of traps is reduced by a factor of 12.3 (17505/1418) from ST and 5.4 (7672/1418) from TDS.

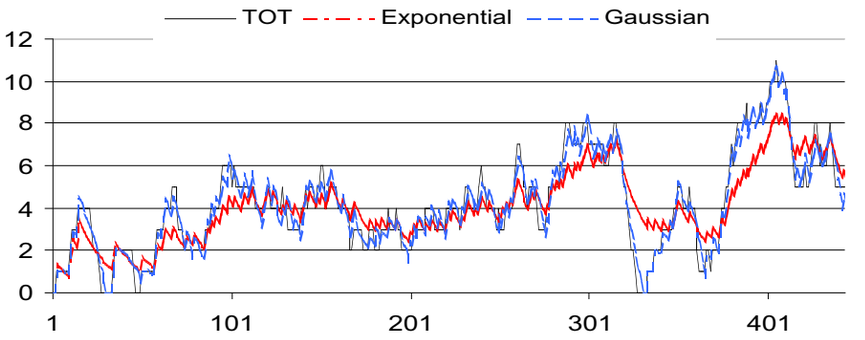
Fig. 1. Comparison of Algorithms



We also examined the behavior of other related algorithms. Roughly, the TOT algorithm can be viewed as a rectangular window filtering of the binary sequence of a monitored variable (1 if sample value over threshold, 0 if below threshold). In an effort to smooth the result, we also tried two other types of window filtering. One is the exponential forgetting filtering, which uses an infinite length window of all available data by putting exponentially attenuated weights on historical samples. The other is a Gaussian shaped window, which has same length as in the TOT but puts more weight on most recent sample and less weight on the past samples. Similar approaches have been used in [1].

The results of these three window filtering are compared in Figure 2. It shows the behavior of a single rule for a single element. The X-axis shows the time, and the Y-axis shows the number of bad polls in an analysis window of width 12. As is shown, these two window alternatives did not improve the TOT algorithm in terms of smoothness and latency. In fact, these two window alternatives gave less smooth results due to non-integer operations. Because these alternative filters are more difficult to explain, and gave no better result, we chose to stick with the simple TOT rectangular filter.

Fig. 2. Comparison of Alternative Window Filtering



7 Conclusion and Future Work

The time over threshold detection algorithm does a good job of reducing the number of alarms raised, and therefore the number of traps that must be processed by an NMS. Yet it does not reduce the ability to detect problems. By far the majority of isolated spikes are transient conditions, which are not indicative of problems. We have implemented algorithms that dynamically determine thresholds based on a statistical analysis of historical behavior. While that work is not covered in this paper, we believe it to be a fruitful area for further research. We certainly have not examined all of the noise reduction algorithms that might be applied to performance and fault management. We believe that any algorithms proposed must be evaluated against real world data such as used in this work.

Of course the true test of any problem detection system is field experience in detecting real problems in real networks, systems, and applications. Our experience with Live Health to date in these real world networks indicates that the basic Time Over Threshold algorithm is effective at reducing noise.

8 Figures

Fig. 3. Typical Variation, Outbound Utilization on a 128 Kbit/sec Link

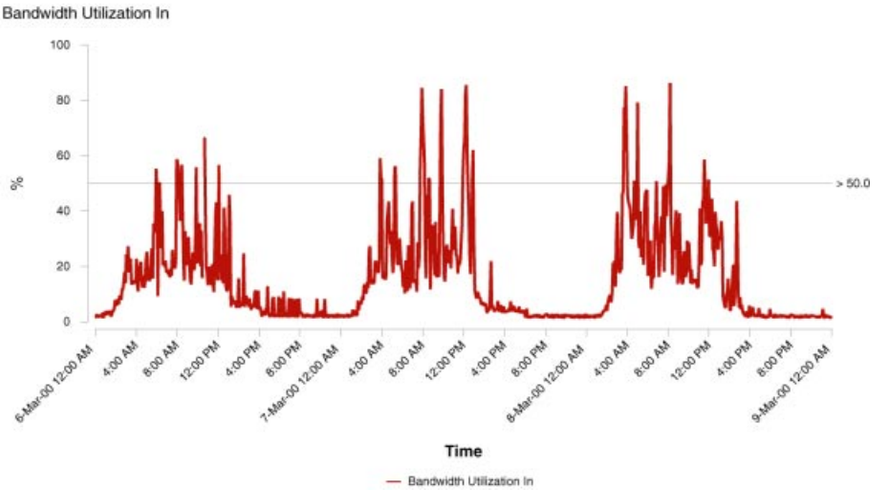


Fig. 4. Errors on a 64Kbit/sec WAN Link

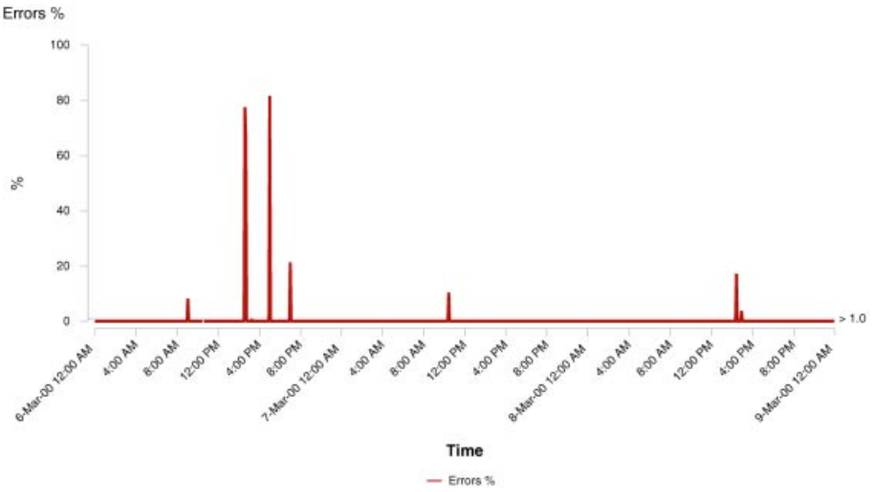


Fig. 5. Outbound Utilization of a 128 Kbit/sec WAN Link

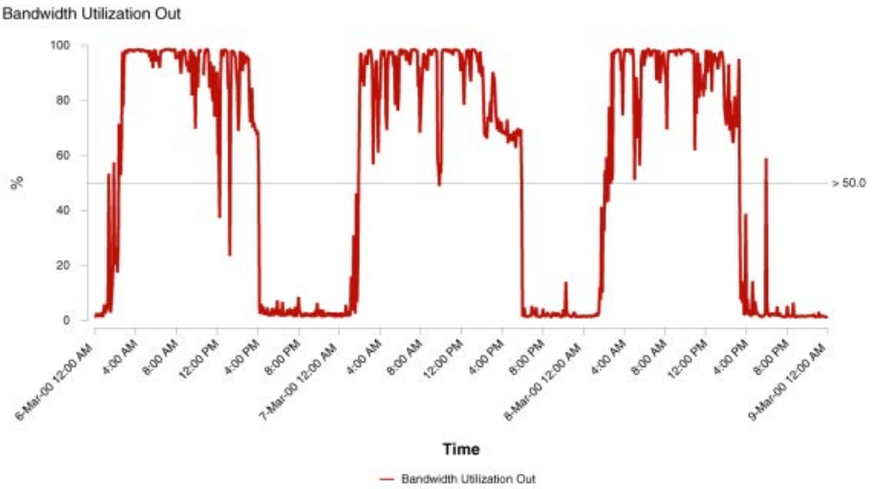


Fig. 6. CPU Utilization too High Alarm that Causes TDS Babbling

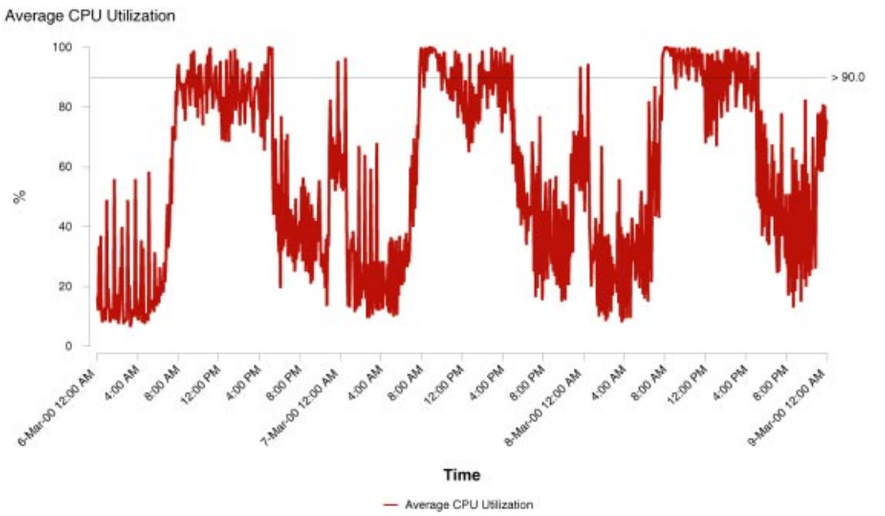
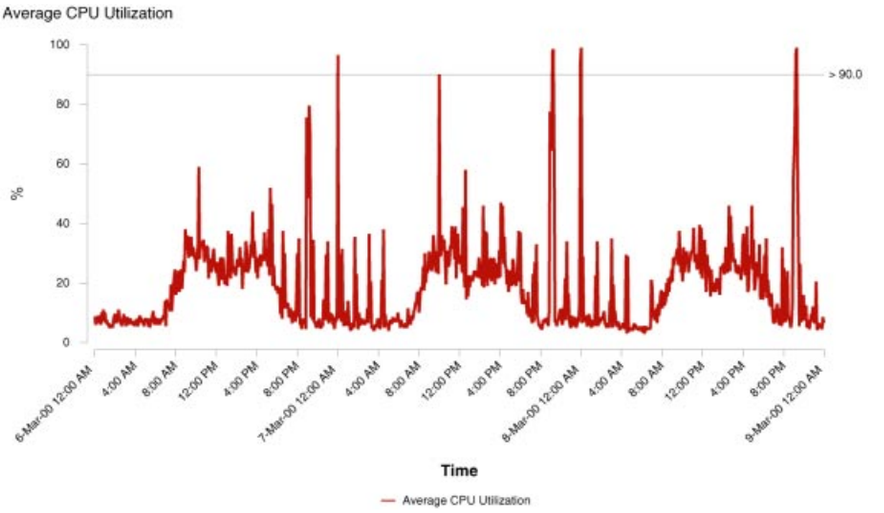


Fig. 7. CPU Utilization of a Unix Server



References

[1] Bondavalli, A., Chiaradonna, S., Di Giandomenico, F., Grandoni, F., Threshold-Based Mechanisms to Discriminate Transient from Intermittent Faults , *IEEE Trans. on Computers*, v 49, no 3, Mar 2000, pp. 230-245.

[2] Hellerstein, J. Zhang, F., and Shahabuddin, P., An Approach to Predictive Detection for Service Management , *Integrated Network Management VI,* Edited by Sloman, M., Mazumdar, S., and Lupu, E., 1999, IEEE Publishing, pp. 309-322.

[3] Huntington-Lee, J., Terplan, K., and Gibson, J., *HP OpenView: A Managers Guide*, McGraw-Hill, New York, NY, 1997, pp. 137-9.

[4] Lelend, W., Taqqu, M., Willinger, W., Wilson, D., On the Self-Similar Nature of Ethernet Traffic (ExtendedVersion) , *IEEE/ACM Trans. on Networking* , v. 2, no.1, Feb 1994, pp.1-15.

[5] ISO/IEC 10164-11:1994 *Information Technology — Open Systems Interconnection — Systems Management: Metric Objects and Attributes.*

[6] Maggiora, P., Elliott, C., Pavone, R., Phelps, K., and Thompson, J., *Performance and Fault Management*, Cisco Press, Indianapolis, IN, 2000, pp. 91-97.

[7] Thottan, M., and Ji, C., Adaptive Thresholding for Proactive Network Problem Detection , *Third IEEE International Workshop on Systems Management*, Newport, RI, Apr 22-24, 1998, pp. 108-116.