Tracking Probabilistic Correlation of Monitoring Data for Fault Detection in Complex Systems

Zhen Guo[*]
Dept. of Electrical & Computer Engineering
New Jersey Institute of Technology
Newark, NJ 07102
Email: zg2@njit.edu

Guofei Jiang, Haifeng Chen, Kenji Yoshihira
Robust and Secure System Group
NEC Laboratories America
Princeton, NJ 08540
Email: {gfj,haifeng,kenji}@nec-labs.com

Abstract

Due to their growing complexity, it becomes extremely difficult to detect and isolate faults in complex systems. While large amount of monitoring data can be collected from such systems for fault analysis, one challenge is how to correlate the data effectively across distributed systems and observation time. Much of the internal monitoring data reacts to the volume of user requests accordingly when user requests flow through distributed systems. In this paper, we use Gaussian mixture models to characterize probabilistic correlation between flow-intensities measured at multiple points. A novel algorithm derived from Expectation-Maximization (EM) algorithm is proposed to learn the "likely" boundary of normal data relationship, which is further used as an oracle in anomaly detection. Our recursive algorithm can adaptively estimate the boundary of dynamic data relationship and detect faults in real time. Our approach is tested in a real system with injected faults and the results demonstrate its feasibility.

1. Introduction

The prevalence of Internet services such as Google.com and Amazon.com has dramatically changed our daily life. Due to the growing scale and complexity of the information systems underlying Internet services, there are unprecedented needs to ensure their operational reliability and availability. Minutes of service downtime can lead to severe revenue loss and users' dissatisfaction. While a large Internet service usually consists of thousands of components such as application software, operating systems, databases, servers and networking devices, a single fault could cause the whole service down. While building a fault-free system of such complexity is unrealistic, it is very desirable to have effective fault detection and isolation methods in operational system management. Studies [1] have shown that the time taken to detect and isolate faults is a major contributor to *mean time to repair (MTTR),* defined to be the average amount of time required to resolve problems. For example, on November 1, 2005, trading on the Tokyo stock market was suspended for four hours after its IT systems failed. Therefore, effective fault detection and isolation is critical for improving the reliability and availability of mission critical systems.

Meanwhile, large amount of monitoring data can be collected from distributed systems for fault analysis such as software log files, system audit events and network traffic statistics. In fact this monitoring data can be regarded as the observables of the internal states of dynamic systems. However, given the distributed nature of complex information systems, evidence of fault occurrence is often scattered in various monitoring data collected across distributed systems and observation time. Therefore, one challenge is how to correlate the monitoring data effectively for fault analysis. Internet services have large number of user requests everyday and much of the monitoring data reacts to the volume of user requests accordingly when user requests flow through the system. For example, network traffic volume and number of web server log entries change in accordance with the volume of user requests. Therefore we can model the correlation between various monitoring data collected at multiple points for fault analysis. Recently we proposed to model and track transaction flow dynamics for fault detection in complex systems [2]. We calculated the *flow intensity* from monitoring data to measure the intensity with which various monitoring data responds to the volume of user requests. Segments of distributed

[*] The work was performed when the author worked with Dr. Guofei Jiang as a summer Intern at NEC Labs.

systems are regarded as input-output dynamic systems and linear regression models are used to characterize the correlation among various flow intensities measured at multiple points across the system. Further these models are used as oracles in model-based fault detection and isolation. While many of such correlations could have linear property, some other measurements may only have probabilistic relationship among them due to many uncertainties embedded in systems such as caching. In this paper, we extend our previous work and propose to model and track such relationships with Gaussian Mixture Models (GMMs) for fault detection.

Tracking probabilistic relationship for fault analysis is essentially an anomaly detection problem. In general, an underlying model is learned or extracted from given data samples and then a new data point is evaluated against the model to derive its deviation score. In practice, run-time faults are rare in operational environments and they also manifest themselves differently in various situations. Therefore it's usually difficult to characterize faults directly in complex systems. Conversely, we have sufficient monitoring data collected from endless business transactions to model normal system behavior. In this paper, a finite GMM is employed to describe the probability distribution of flow intensities calculated from monitoring data. A new variant of EM algorithm [3][4] is used to learn the parameters of Gaussian mixtures and further estimate a boundary of normal data distribution. In fact we use this boundary to distinguish outliers from normal data points, i.e., detect the outliers with extremely low probability density. Our fault detection algorithm is derived from an online recursive EM algorithm so that it can adapt to system and load changes. In addition, we propose a procedure to automatically search and validate probabilistic relationships between each pair of monitoring data. Our approach is tested in a real system with injected faults and the results demonstrate its feasibility.

2. Related work

There is much work about fault detection and isolation in telecommunication network management. Yemini et al. [5] proposed a "Codebook" approach for high speed and robust event correlation. Chao et al. [6] developed an automated fault diagnosis system using hierarchical reasoning and alarm correlation. Recently, Benveniste et al.[7] employed a net unfolding approach originating from the Petri net research for distributed fault diagnosis. All these methods collect and correlate events to locate faults based on known dependency knowledge between faults and symptoms. While this knowledge can be derived from network topology for telecommunication network, it is very difficult to obtain such kind of knowledge in large and complex information systems.

The Berkeley Recovery-Oriented Computing (ROC) group 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 [1]. However, with regard to huge volume of user visits, it is difficult to monitor, collect and analyze the trace of each individual user request. Based on commonly available monitoring data such as log files, recently we [2] proposed to model and track transaction flow dynamics for fault detection in complex systems. However, we only employed linear regression models to characterize dynamic relationships among flow intensities measured at multiple points. In this paper we propose to use Gaussian mixtures to model probabilistic relationships of monitoring data and further use these models for fault detection. As illustrated in our experiments, due to system dynamics and uncertainties, we believe that some measurements may only have weak and probabilistic correlation rather than linear relationship.

Vaarandi [8] applied clustering algorithms to mine event logs for fault detection. The work first constructs clusters by grouping event logs based on their message characters and then detects failures by tracking anomalous events which do not belong to any existing clusters. Based on Gaussian mixtures, Yamanishi et al. [9] developed an online unsupervised outlier detection engine named Smartsifter and it was used for intrusion detection in computer security. Though Gaussian mixtures are also used to model data relationship in our work, we apply a new variant of recursive EM algorithm that is able to tune the number of clusters dynamically. Meanwhile, we track the probabilistic relationship among monitoring data rather than their real values so that we have a thoroughly different approach for outlier detection.

3. Probabilistic relationship

Many Internet services employ multi-tier system architecture to integrate their components. Typical three-tier architecture is illustrated in Figure 1 and it includes web servers, application servers and database servers. A web server acts as the system interface to handle requests and responses from/to clients. An application server supports specific application and business logic for Internet services. Database servers are the back-end servers for persistent data storage. Much monitoring data can be collected from such systems for fault analysis. For example, the access logs of web server include every HTTP requests from clients. The runtime monitoring data of application servers such as the numbers of threads and EJBs can be monitored via Java Management Extension (JMX) [10]. Many tools are also available to collect network traffic statistics and OS audit data such as CPU and memory usage.

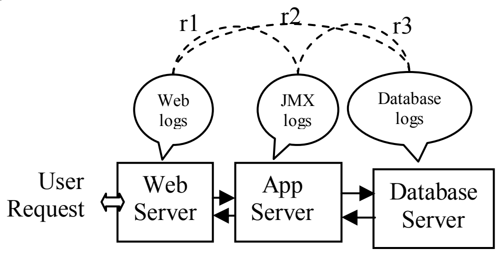

Figure 1. Typical three-tier systems

The rich set of monitoring data collected at multiple points enables us to analyze their correlations across distributed systems. In Figure 1, r1, r2 and r3 indicate three different correlations between related monitoring data. For example, r1 could represent the correlation between the volume of user requests and the number of Java threads running on the application server. In this paper, we select pairs of such measurements, $x(t)$ and $y(t)$, to form a set of two-dimensional variables: $(x(t), y(t))$, which correspond to data points in a 2-D space as shown in Figure 2. As mentioned above, much of monitoring data reacts to the intensity of user requests accordingly when user requests flow through the system. Here we use flow intensity to measure the intensity with which various internal monitoring data responds to the volume of user requests. Since these measurements are affected by the same factor – user loads, we believe that there is much correlation between these measurements. Meanwhile, due to system dynamics and uncertainties, some correlations may only be characterized with probabilistic models.

If we consider a system component as a black box, the correlation between the monitoring data measured at the input and output of the component could well reflect the constraints that the monitored system bears. As an engineered system, the constraints could be imposed by many factors such as hardware capacity, application software logic and system architecture. After some faults occur inside the component, some of such constraints could be violated and we may detect such faults by continuously tracking the correlation between the input and output measurements. In this paper, we use GMMs to approximate the probabilistic correlation between flow intensity measurements. A probability density boundary is determined by tracking the mass characteristics of historical measurements. Since the probability density of a data point inside the boundary is always greater than that outside the boundary, we detect anomalies by determining whether a new data point locates outside of the correlation boundary, i.e., determining whether the probability density $p(x(t), y(t))$ is less than the selected $p_{boundary}$.

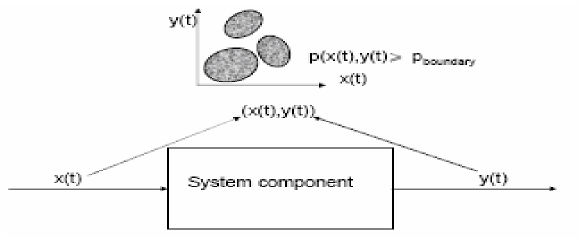

Figure 2. Correlation of monitoring data

4. Gaussian mixture models

Figure 3 shows an example of the correlation between memory and number of Java threads used in the middleware server. The samples are collected from a real system, which will be introduced in Section 8. From the scatter plot in Figure 3, we notice that data points can be clustered together around several centers with compact cluster size. We apply a GMM to characterize such relationship because many flow intensity measurements are likely to follow Gaussian distribution. As mentioned above, the volume of user requests is the major factor that affects the intensity of internal monitoring data. The flow intensity measurements are most likely to respond to the current level of user loads accordingly but are also affected by some uncertainties. Therefore, given a certain volume of user requests, these measurements are likely to follow Gaussian distribution and their mean values could well correspond to the current intensity of workloads. Meanwhile, due to varying workloads, we may need multiple Gaussian distributions to capture the mass characteristics of measurements. This is our early motivation for using GMMs in this work.

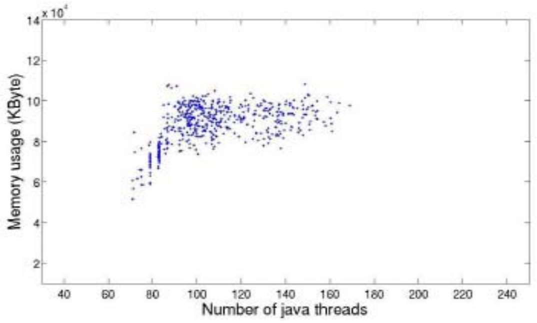

Figure 3. Correlation between number of Java threads and memory usage

In a GMM, we use the following probability density function p to approximate the real data distribution:

$$p(z_i|\theta)=\sum_{j=1}^{G}\alpha_j p_j(z_i\,|\,\mu_j,\Sigma_j)\,. \quad (1)$$

In this paper, we only consider the correlation between each pair of measurements. Though we can use GMMs to correlate high dimensional data, such correlation is not helpful in locating faults. Therefore data points are two-dimensional vectors denoted by, $\{z_i\}=\{(x_i,y_i)\}$, $1\le i\le N$ where N is the number of data samples. In Equation (1), G is the number of mixtures and $\alpha_1,...,\alpha_G$ are the unknown proportions of these mixtures. Thus we have $\sum_{j=1}^{G}\alpha_j=1$. $p_j(z_i\,|\,\mu_j,\Sigma_j)$, denotes the j-th two-dimensional Gaussian distribution with its mean value μ_j and covariance matrix Σ_j, i.e.,

$$p_j(z\,|\,\theta_j)=\frac{1}{2\pi\left|\Sigma_j\right|^{1/2}}\exp\{-\frac{1}{2}(z-\mu_j)'\Sigma_j^{-1}(z-\mu_j)\} \quad (2)$$

Denote the mixture parameter set by $\theta=\{(\alpha_i,u_i,\Sigma_i),$ $1\le i\le G\}$. As shown in Equation (1), the probability density of a data point is a weighted mixture of the G Gaussian distributions. Given data samples, the well known EM algorithm [3] can be used to estimate the optimal parameter set $\hat{\theta}$ that maximally approximates the real data distribution.

5. Model-based fault detection

For two-dimensional Gaussian distributions shown in Equation (2), the *Mahalanobis distance* is defined as $m(z)=((z-\mu)'\Sigma^{-1}(z-\mu))^{1/2}$. For a specific Gaussian distribution $p(z)$, the values of its squared Mahalanobis distance, $m(z)^2$, are approximately Chi-square distributed [11]. A probability density boundary determined by $p(z)$ can be mapped to a corresponding boundary in its Mahalanobis distance. Therefore, we can use the following inequality to determine the correlation boundary in Mahalanobis distance:

$$m(z)^2=(z-\mu)'\Sigma^{-1}(z-\mu)=(z-\mu)'(P\Lambda P')^{-1}(z-\mu)$$
$$=\vec{t}\,'\Lambda^{-1}\vec{t}=\frac{t_1^2}{\lambda_1}+\frac{t_2^2}{\lambda_2}\le\chi^2(\alpha) \quad (3)$$

where λ_1 and λ_2 are the eigenvalues of the covariance matrix Σ, α is the probability coverage in Gaussian distribution and $\vec{t}=(P')^{-1}(z-\mu)=\begin{pmatrix}t_1\\t_2\end{pmatrix};\Lambda=\begin{pmatrix}\lambda_1&0\\0&\lambda_2\end{pmatrix}$.

For example, if 90% of data points have to be included inside the boundary, then we have $\alpha=0.9$. For a selected α of a Gaussian distribution, we can calculate a $\chi^2(\alpha)$ so that $P(m(z)^2\le\chi^2(\alpha))=\alpha$, i.e., the probability coverage α determines an ellipse size $\chi^2(\alpha)$ in Mahalanobis distance with which the ellipse could include 100*α percentage of data points. Note that Inequality (3) represents an ellipse in two-dimension space of monitoring data. Each cluster in the mixture corresponds to an ellipse so that we totally have G ellipses to characterize the probabilistic relationship between monitoring data.

If the parameters of Gaussian distributions, θ , are known and α is given, we can use Inequality (3) to determine the boundary in Mahalanobis distance. Given the real data observations, the question here is how to learn the parameters θ with which the GMM could best approximate the real data distribution. The usual choice for maximizing the *posteriori* estimates of mixture parameters is the well known EM algorithm. Due to system dynamics such as varying workloads and uncertainties such as caching, we use an online and recursive EM algorithm to update models dynamically in operational environments. A new data point is regarded as an outlier if it locates outside of the selected boundary. This section proposes a fault detection algorithm which includes the recursive EM algorithm as well as a method from outlier detection. The recursive EM algorithm [4] is an online discounting variant of EM algorithm. In real time applications, a recursive EM algorithm is better than the classic EM in the sense that the parameters are updated online with a forgetting factor degrading the influence of out-of-date samples. Thus a recursive EM algorithm is capable of adapting to system dynamics in real time.

The EM algorithm is an iterative procedure that searches for the parameters that maximize a log-likelihood function as shown in Equation (4). It consists of the following two basic steps:

$$\textit{E-Step: } Q(\theta,\hat{\theta}_{k-1})=E_l(\log p(Z,l\,|\,\theta)\,|\,Z,\hat{\theta}_{k-1})\,.$$
$$\textit{M-Step: } \hat{\theta}_k=\arg\max_{\theta}(Q(\theta,\hat{\theta}_{k-1}))\,. \quad (4)$$

where Z is the set of data samples, and $l=\{l^{(1)},l^{(2)},...,l^{(G)})$ is a binary vector of G labels indicating which cluster a data sample belongs to, for example if $l^{(1)}=1,l^{(2)}=...=l^{(G)}=0$ it means that this sample belongs to cluster 1. For GMMs, the EM algorithm is based on the observations Z as incomplete data and the missing part is the information of labels. The complete log likelihood is $\log p(Z,l\,|\,\theta)$, from which we could estimate θ if the complete data $\{Z,l\}$ is given. The first step, referred to as *E-Step*, computes the conditional log-likelihood given the observation Z and the parameter estimate $\hat{\theta}_{k-1}$. The second step is to search the parameters that maximize the log-likelihood.

Rather than maximizing $Q(\theta, \hat{\theta}_{k-1})$ in the standard EM algorithm, we maximize the following criteria in the recursive EM algorithm [4].

$$\hat{\theta}_k = \arg\max_{\theta} J; J = Q(\theta, \hat{\theta}_{k-1}) - \mathrm{V}(\alpha)$$
$$= E_l(\log p(Z, l \mid \theta) \mid Z, \hat{\theta}_{k-1}) + \log \prod_{j=1}^{G} \alpha_j^{c_j} \tag{5}$$

where $V(\alpha)$ is introduced as a penalty function to remove unnecessary clusters in the mixtures. Here $V(\alpha) \propto -\sum_{j=1}^{G} c_j \log \alpha_j = -\log \prod_{j=1}^{G} \alpha_j^{c_j}$ is the negative logarithm of a Dirichlet prior [12] and $\alpha = \{\alpha_1, ..., \alpha_G\}$, $\sum_{j=1}^{G} \alpha_j = 1$. It is straightforward to see that the penalty function always decreases with fewer clusters. $c_j = -D/2$ where D represents the number of parameters per cluster. For the Maximum A Posteriori (MAP) solution, using Lagrange optimization method, we can have

$$\frac{\partial}{\partial \alpha_j}(E_l(\log p(Z, l \mid \theta) \mid Z, \hat{\theta}_{k-1})$$
$$+ \log \prod_{j=1}^{G} \alpha_j^{c_j} + \lambda(\sum_{j=1}^{G} \alpha_j - 1)) = 0 \tag{6}$$

where λ is the Lagrange multiplier. Based on Equation (6), the following recursive equations are obtained:

$$o_j^t(z^{t+1}) = \frac{\alpha_j^t p_j(z^{t+1} \mid \hat{\theta}_j^{\,t})}{p(z^{t+1} \mid \hat{\theta}^t)};$$
$$\alpha_j^{t+1} = \alpha_j^t + \rho\left(\frac{o_j^t(z^{t+1})}{1 - Gc_T} - \alpha_j^t\right) - \rho \frac{c_T}{1 - Gc_T} \tag{7}$$
$$= \rho\left(\frac{o_j^t(z^{t+1}) - c_T}{1 - Gc_T}\right) + (1 - \rho)\alpha_j^t$$

where $c_T = D/(2T)$ and T is a constant integer which should be large enough to make sure that $Gc_T < 1$. z^{t+1} is the new data point collected at sampling time $t+1$. $\rho = 1/T$ is a fixed forgetting factor used to reduce the influence of out-of-date samples. After the new weights of mixtures α_j^{t+1} are calculated, the online algorithm checks whether there are unnecessary clusters in the mixtures or not and removes those clusters with $\alpha_j^{t+1} < 0$. As mentioned earlier, this mechanism of discarding unnecessary clusters is achieved by introducing the penalty function. Then the rest of the parameters can be updated with the following equations:

$$\mu_j^{t+1} = \mu_j^{\,t} + w\delta, \ \Sigma_j^{t+1} = \Sigma_j^t + w(\delta\delta^T - \Sigma_j^t),$$
$$\text{where } w = \rho \frac{o_j^t(z^{t+1})}{\alpha_j^t}, \delta = (z^{t+1} - \mu_j^{\,t}). \tag{8}$$

For each iteration, these parameters are updated with a new data point. The boundaries in Mahalanobis distance, which are ellipses as illustrated in Inequality (3), are also slightly tuned at each step to update the probabilistic relationship of monitoring data. Here let us consider the correlation example between number of Java threads and memory usage, as shown in Figure 3. After 250 data points, Figure 4 illustrates the updated probabilistic relationship resulting from the above recursive EM algorithm. Note that these ellipses are determined by Inequality (3) and have 99% probability coverage. All points on the elliptical boundary of a same cluster have the same probability density. For each Gaussian distribution $j \in [1, G]$, we denote the probability density of data points on its boundary as $p(b_{t,j} \mid \theta_j)$ where $b_{t,j}$ represents any data point on the boundary of the j-th Gaussian distribution at time t. Note that $b_{t,j}$ doesn't have to be the data point observed from real systems. Instead it is a data point in the learned continuous GMM.

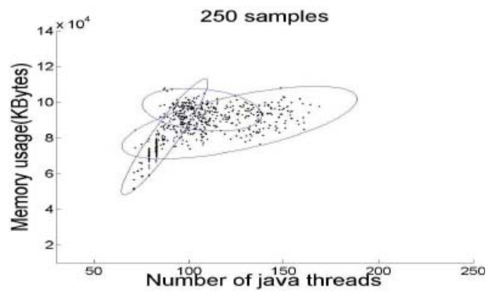

Figure 4. Boundaries of probabilistic relationship

As briefly discussed in Section 3, a fault occurrence inside the monitored component could affect the correlation between monitoring data measured at the input and output of that component. Therefore we can expect to detect such a fault by tracking whether the probability density of the new measurement is less than the probability density on the boundaries. If a data point z_t is included by at least one ellipse, then we know that $\exists j \in [1, G],\ p(z_t \mid \theta_j) - p(b_{t,j} \mid \theta_j) \geq 0$. If a data point z_t locates outside of any ellipses, then we know that $\forall j \in [1, G],\ p(z_t \mid \theta_j) - p(b_{t,j} \mid \theta_j) < 0$. Thus we can use the following criteria to determine outliers: For $j \in [1, G]$, if $\mathrm{Max}_j\ (p(z_t \mid \theta_j) - p(b_{t,j} \mid \theta_j)) \geq 0$, we consider this data point z_t as a normal observation; otherwise, we regard the data point z_t as an outlier. Since the probability density of a data point decreases exponentially with its distance to the center, the probability density of outliers are extremely low. Therefore, it is more convenient to use the logarithms of probability densities $p(z_t \mid \theta_j)$ and $p(b_{t,j} \mid \theta_j)$ to present the residuals:

$$R(t) = \underset{j}{\text{Max}}(\log p(z_t \mid \theta_j) - \log p(b_{t,j} \mid \theta_j)),\ j \in [1, G] \quad (9)$$

Based on the above discussion, we conclude that if a data point locates inside of at least one ellipse, we have $R(t) \geq 0$; otherwise for outliers, we have $R(t) < 0$. Figure 5 shows the residual generated from the probabilistic relationship between number of Java threads and memory usage. In this example, we injected a "Busy loop" fault into the application software running on the middleware by modifying its source code. Details about this fault will be introduced in Section 8.

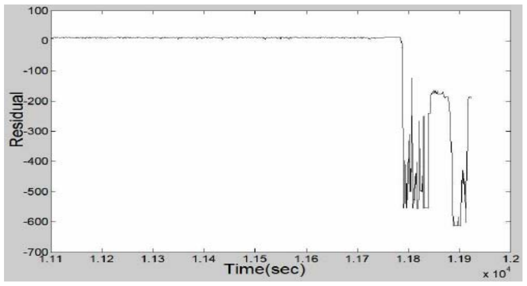

Figure 5. Residual changes in a faulty case

It is straightforward to see that the residual has a significant drop after the fault was injected. An online tracking and detection algorithm is shown in Figure 6. Because Gaussian distribution has a large tail, the size of ellipses would become extremely large as they approach to have 100% probability coverage. From one side, we expect the ellipses to be fairly compact so as to reduce miss-detections, and on the other side we also expect the probability coverage to be high so as to reduce false positives. In this paper, we choose the ellipses to have 99% probability coverage so as to have reasonable compact size. That means our approach could result in 1% false positives of all data points. In 24*7*365 operational environments, this false positive rate could lead to thousands of false positive cases. Thus our algorithm only generates an alert after three consecutive data points are detected as outliers. After a fault occurs, it's often persistent (i.e., not transient) before it is eventually resolved by operators. Thus faults are likely to cause many consecutive outliers, as shown in Figure 5 and in Section 8. Meanwhile, since noise resulting from uncertainties is usually independent, the possibility to have three consecutive false positives is as low as 10^{-6} .

```
Algorithm 5.1
Input: $z_t$ , initial parameters $\theta(0)$
Output: abnormal residual alert
For each time step t,
1. Compute $p(z_t \mid \theta_j)$ and $p(b_{t,j} \mid \theta_j)$ for each
   $j \in [1, G]$ using the parameters $\theta_j(t-1)$ ;
2. Compute the residual
   $R(t) = \text{Max}_j(\log p(z_t \mid \theta_j) - \log p(b_{t,j} \mid \theta_j))$ ;
3. If $R(t) \geq 0$ , then
      Update $\theta(t) = \{\theta_j(t)\}, j \in [1, G]$ using recursive EM;
   else if $R(t-1) < 0$ & $R(t-2) < 0$ , then
      Generate an alert.
```

Figure 6. Online tracking and detection algorithm

6. Residual correlation

We can generate n residuals if we can model n such relationships among monitoring data. In a normal situation, these residuals reflect the error resulting from modeling and their absolute values are usually very small. If some fault occurs inside the system, some of these n relationships could be affected and their residuals could significantly drop. Therefore, we could detect such a fault by tracking the change of these residuals. In complex systems, faults are very diverse and could include various software bugs, hardware problems and operator mistakes.

	C1	C2	C3	…	Cm
R1	1	1	0	…	0
R2	1	0	0	…	0
…	…	…	…	…	…
Rn	0	1	1	…	1

Figure 7. Residual correlation matrix

While some faults may cause several residuals to be abnormal, these faults may have no impact on other residuals. Therefore whether a specific fault can be detected is dependent on its impact on these residuals. If we have many of such residuals, we would expect to detect a wide class of faults in the system. Since the residuals are calculated directly from the probabilistic relationship, they can reflect how healthy the monitored components are. Based on the dependencies between residuals and their monitoring components, we can develop a correlation matrix as shown in Figure 7. The columns represent the components of the monitored systems while the rows indicate the residuals. We define the matrix elements in Boolean type: $V_{ij} = 1$ means that the component C_j is being monitored by the residual R_i, and 0 otherwise.

Algorithm 5.1 keeps tracking the modeled relationships and generating residuals. For each residual at time step t, we use a Boolean variable, $r_i(t)$, to indicate whether the residual $R_i(t)$ is abnormal, i.e., $R_i(t) < 0$. We let $r_i(t) = 1$ if the residual $R_i(t)$ is abnormal, otherwise $r_i(t) = 0$. Therefore, at each time step t, we can have an observation vector: $O(t) = [r_1(t), r_2(t), ..., r_n(t)]^T$. Based on the dependency knowledge described in Figure 7, we can use the following Jaccard coefficient to locate the faulty component most likely:

$$C_f = \arg\max_j \frac{\sum_{i=1}^{n} r_i(t) \cap V_{ij}}{\sum_{i=1}^{n} r_i(t) \cup V_{ij}} \tag{10}$$

Basically Equation (10) identifies the component, which is maximally correlated with the current residual observation $O(t)$, as the faulty component. This fault isolation mechanism assumes that the faulty component would cause most of its related residuals to be abnormal. In addition, rather than using Equation (10) to determine one faulty component, the correlation matrix in Figure 7 could enable operators to narrow down the set of possible faulty components according to the current residual observation $O(t)$ and reduce their time in pinpointing the fault.

7. Automated model search and validation

In Section 4, we analyzed how to automatically build a probabilistic relationship between two measurements based on GMMs. In complex systems, we may collect many pairs of measurements. However, the question is how to determine whether the real data distribution fits the Gaussian distribution or not. The correlation between some measurements may not be well characterized with GMMs if their real data distribution doesn't follow Gaussian distribution.

In this paper, we use the normalized difference between the estimated and empirical data distribution to measure how well the learned model fits the real data distribution. For convenience, we divide each cluster into several segments with equal area as shown in Figure 8. For each segment, we calculate the estimated probability based on the learned model as well as the empirical probability based on the real data distribution. Figure 8 illustrates the segmentation of a cluster, where S is the area of the ellipse with 99% probability coverage. Sharing the same center with that ellipse, $M-1$ smaller ellipses can be drawn with area size equal to $S/M, ..., kS/M, ..., (M-1)*S/M$ respectively. Here the *segments* refer to the $M-1$ "elliptic rings" as well as the smallest ellipse in the center. The estimated probability of a segment k is defined to be the probability of segment k using the estimated GMM while the empirical probability of segment k is the ratio between the number of data points in segment k and the total number of data points N .

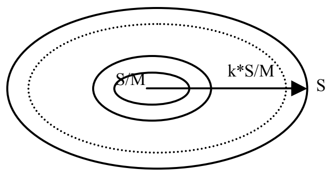

Figure 8. Segments of a cluster

In this work, we set the boundary of the squared Mahalanobis distance to 9.210, which corresponds to 99% probability coverage in data distribution, i.e., $\chi^2(0.99) = 9.210$. To make M segments equally sized, the ellipses chosen to separate segments have the squared Mahalanobis distances equal to $9.210/M, ...,$ $9.210k/M, ..., 9.210$ respectively and $1 \le k \le M$. Figure 9 shows an example in comparison of the estimated and empirical probability of each segment, which are calculated from the example shown in Figure 3. In this example, we divide each cluster into 6 segments, i.e., *M*=6. From the figure, we observe that the real data distribution fits into the learned GMM well though the two probability distributions have some difference. We use the following equation to calculate a normalized fitness score for model evaluation:

$$F(\theta) = [1 - \frac{\sum_{k=1}^{M} (p_{real}(k) - \hat{p}_{est}(k))^2}{\sum_{k=1}^{M} (p_{real}(k) - \frac{1}{M})^2}] \times 100 \tag{11}$$

where $p_{real}(k)$ and $\hat{p}_{est}(k)$ respectively, represents the empirical probability and estimated probability of data points in *k*-th segment. $1/M$ is the mean of probabilities in all segments, i.e., the probability of uniform distribution. High fitness score indicates that the real data distribution is well characterized by the estimated GMM. Note that the upper bound of the fitness score in Equation (11) is 100.

In practice, we do not have prior knowledge about which measurements may have probabilistic relationships in complex systems. Here we propose an approach to automatically search and validate such

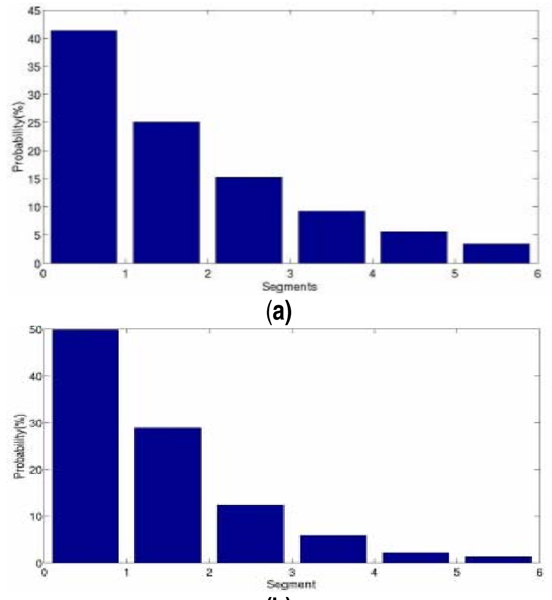

Figure 9. Probability in segments (a) Estimated probability (b) Empirical probability

relationships among monitoring data. We try any combination of two measurements to build a model first, and then apply sequentially testing to validate this model with new data points. We use $F_v(\theta)$ to denote the fitness score that is calculated with data points collected during the v-th time window. A threshold $\tilde{F}$ is chosen to determine whether the model fits the real data distribution or not in the following piecewise function:

$$f(F_v(\theta)) = \begin{cases} 1 & \text{if } F_v(\theta) > \tilde{F} \\ 0 & \text{if } F_v(\theta) < \tilde{F} \end{cases} \qquad (12)$$

After receiving the monitoring data for h of such time windows (with a fixed window size), we can calculate a confidence score, $C_h(\theta)$, with the following equation:

$$C_h(\theta) = \frac{\sum_{v=1}^{h} f(F_v(\theta))}{h} = \frac{C_{h-1}(\theta)\cdot(h-1) + f(F_h(\theta))}{h} \qquad (13)$$

Our automated model search and validation procedure starts to build a model for any pair of measurements first and then incrementally validates these models with new data points. After a time period, if the confidence score of a specific relationship is less than a selected threshold, the model for this relationship is considered to be invalid and we abandon this model. Meanwhile, we continue to test the set of good models and use these models as oracles in fault detection. Note that a robust model is more credible in fault detection than those models with low confidence scores. In an operational environment like Internet services, we always have endless new incoming data for model sequential testing.

8. Fault detection experiments

Our test bed system has typical three-tier system architecture as shown in Figure 1. The system includes Apache web server, JBoss application server and MySQL database server. The application software running on the system is Pet Store [13], which is a blueprint J2EE application. Pet Store has 27 Enterprise Java Beans (EJBs), some Java Server Pages (JSPs) and Java Servlets.

Just like other Internet services, users can go to the Pet Store website to buy various pets. A client emulator has been developed to generate a large class of different user scenarios and workload patterns. Various user actions such as browsing items, searching items, account login, adding an item to a shopping-cart, payment and checkout are included in our workloads. Certain randomness of user behaviours is also considered in the workload. The time between two actions is randomly selected from a reasonable range. Note that workloads are dynamically generated with much randomness and variance so that we never get a similar workload twice in our experiments. The magnitude of workloads is between 0 and 100 user requests per second.

As discussed before, various faults could affect complex information systems in very different ways. In this section, we inject a list of faults into the system and then use the collected monitoring data to compute residuals. Further we check whether these residuals become abnormal in the faulty cases to verify the feasibility of our approach. Note that our approach is not specifically designed to detect these faults and here we just use these known faults for validation purpose. In addition, the effectiveness of our approach is also dependent on the granularity of available monitoring data. In our future work, we will need a large pool of injected faults to quantitatively verify the performance of our approach. The following experiments are only designed to illustrate the feasibility of our approach. For convenience, in the following experiments, we use R1, R2 and R3, respectively, to represent the following three data relationships: Java thread-CPU usage; SQL queries-Memory usage; Web requests-Memory usage. Here we assume that the clocks of different monitoring points are synchronized.

A. Busy Loop

Infinite loop is a notorious software bug that could be made by novices as well as experienced programmers. Typically, unexpected behaviour of a terminating condition can cause this problem. Here we use a general ``busy loop'' fault to simulate this type of faults. The process injected with ``busy loop'' faults will enter a busy loop procedure for a period of time. Figure 10 shows the residuals resulting from busy loop faults. In this experiment, we choose one specific user request to inject this failure by modifying source code. Whenever this specific request is submitted, it will enter a busy loop. Figure 10 shows that all three residuals have strong evidence of this fault's occurrence. The busy loop fault consumes high percentage of CPU power so that it blocks the execution of other user requests in the application server. Therefore the memory usage is not well correlated with the number of HTTP and SQL requests after the fault occurs. This explains why the relationships R2 and R3 are also affected.

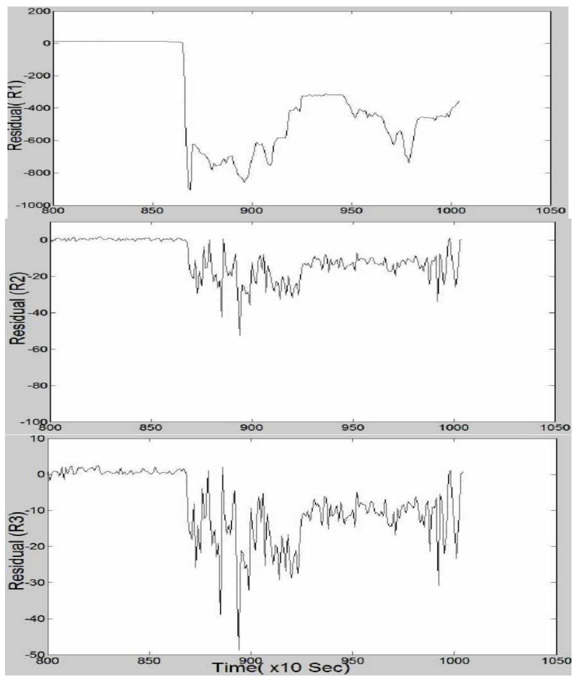

Figure 10. Residuals in a busy loop fault

B. Memory Leaking

Memory leaking is a common software bug with which a program repeatedly allocates heap memory to an object but never releases it. The accumulation of leaked memory may eventually exhaust all the memory available. A program with memory leaking bug could run correctly for a long period of time before it eventually causes something bad to happen. A memory leaking bug may not manifest itself in the same way all the time. Though Java supports garbage collection, memory leaking could still happen because allocated objects which are no longer needed can remain reachable from useful and long-lived objects, and thus they are not garbage collected. Figure 11 illustrates the residuals of three relationships resulting from memory leaking faults. In this experiment, we injected this failure into the "ShoppingCart" EJB of the Pet Store application by repeatedly allocating some heap memory and make it reachable from a long-lived object. Memory leaking eventually leads to heavy operation of JVM garbage collection.

Essentially this abnormal operation affects the probabilistic correlation characterized in those three models. For example, many requests and processes are delayed to yield the garbage collection during the time of this operation. Therefore, all of three residuals show strong signal revealing the existence of a fault. Compared with the continuous distorted segments of residuals in busy loop faults shown in Figure 11, memory leaking faults only lead to discrete downward "spikes" periodically. This is because the heavy garbage collection operations only occur once in a while. Each time after an operation is completed, the system goes back to the normal situation and therefore the downward spikes appear periodically.

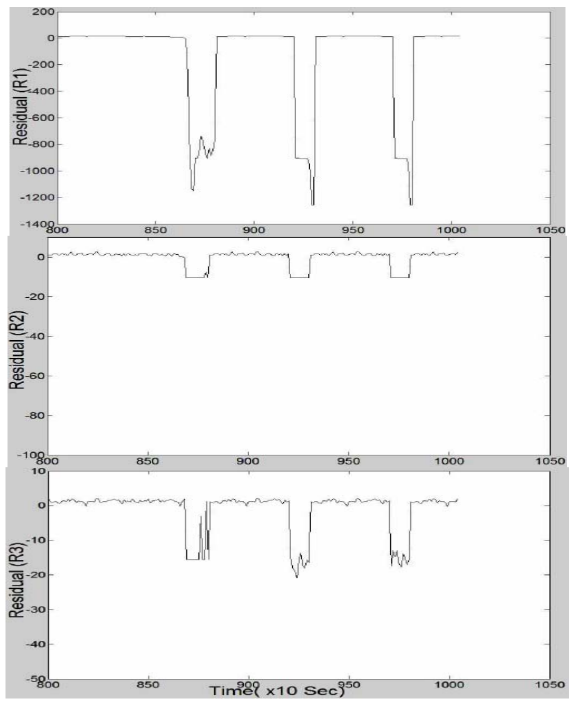

Figure 11. Residuals in a memory leaking fault

C. Deadlock

Deadlocks are a common problem in multi-threaded processing where several processes share a mutually exclusive resource, commonly protected by a lock. A deadlock could cause multiple participating processes to enter an endless waiting loop. Here we use a "waiting loop" in our experiments to simulate the impact of deadlock problems. Compared to the experiment of "busy loop" faults, the only difference is that we changed the "busy loop logic" to the "waiting loop logic" in the related source code. For a specific user request, it will enter a waiting loop that randomly lasts from 4 to 12 seconds with no operation in our experiment. In the busy loop fault case, all three residuals become abnormal after the fault occurs. Conversely, as shown in Figure 12, only R3 becomes abnormal in the deadlock case while R1 and R2 seem to be normal. This is because the "waiting loop" fault only affects the total number of the injected requests but has no impacts on other requests at all.

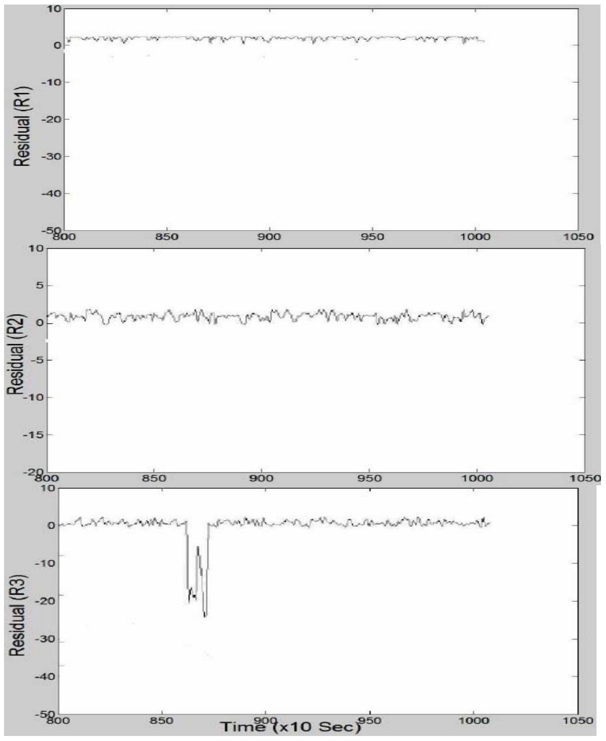

Figure 12. Residuals in deadlock faults

9. Discussions

In this paper, we report our preliminary results of tracking probabilistic relationship of monitoring data for fault detection in complex systems. Note that the above experimental results only demonstrate the feasibility of our approach though all experiments are repeated. In our future work, we need to design a wide class of fault injection methods and run many experiments to quantitatively verify the robustness and effectiveness of our approach such as false positive and negative rates in fault detection.

References

[1] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox and E. Brewer, "Path-based failure and evolution management", in *1st USENIX Symposium on Networked Systems Design and Implementation (NSDI'04)*, San Francisco, CA, March 2004.

[2] G. Jiang, H. Chen and K. Yoshihira, "Discovering likely invariants of distributed transaction systems for autonomic system management", *The 3rd International Conference on Autonomic Computing*, Dublin, Ireland, June, 2006.

[3] J. Bilmes, "A gentle tutorial of the EM algorithm and its application to parameter estimation for Gaussian mixture and hidden Markov models," *Technical Report*, U.C. Berkeley, April, 1998.

[4] Z. Zivkovic and F. Heijden, "Recursive unsupervised learning of finite mixture models", *IEEE Transactions on Pattern Analysis and Machine Intelligence,* vol. 26, no. 5, pp. 651–656, May, 2004.

[5] A. Yemini and S. Kliger, "High speed and robust event correlation", *IEEE Communication Magazine*, vol. 34, no. 5, pp. 82-90, May, 1996.

[6] C. Chao, D. Yang and A. Liu, "An automated fault diagnosis system using hierarchical reasoning and alarm correlation", *Journal of Network and Systems Management,* vol.9, no.2, pp. 183-202, June, 2001.

[7] A. Benveniste, E. Fabre, C. Jard and S. Haar, "Diagnosis of asynchronous discrete event systems, a net unfolding approach", *IEEE Transactions on Automatic Control*, vol. 48, no. 5, pp. 714-727, May, 2003.

[8] R. Vaarandi, "A data clustering algorithm for mining patterns from event logs," in *Proc. of 2003 IEEE Workshop on IP Operations & Management (IPOM2003)*, Kansas City, Missouri, October, 2003.

[9] K. Yamanishi, J. Takeuchi, G. Williams and P. Milne, "On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms", in *Proceedings of the sixth ACM international conference on Knowledge discovery and data mining*, pp. 320-324, Boston, MA, August, 2000.

[10] http://java.sun.com/products/JavaMangement/

[11] P. Filzmoser, "A multivariate outlier detection method", in *Proc. of the Seventh International Conference on Computer Data Analysis and Modeling,* volume 1*,* pp. 18-22, Minsk, Belarus, 2004.

[12] A. Gelman, J.B. Carlin, H.S. Stern and D.B. Rubin, *Bayesian data analysis*, Chapter 16, Chapman and Hall, 1995.

[13] Http://java.sun.com/deveoper/releases/petstore/.