Modeling and Tracking of Transaction Flow Dynamics for Fault Detection in Complex Systems

Guofei Jiang, *Member*, *IEEE*, Haifeng Chen, and Kenji Yoshihira

Abstract—With the prevalence of Internet services and the increase of their complexity, there is a growing need to improve their operational reliability and availability. While a large amount of monitoring data can be collected from systems for fault analysis, it is hard to correlate this data effectively across distributed systems and observation time. In this paper, we analyze the mass characteristics of user requests and propose a novel approach to model and track transaction flow dynamics for fault detection in complex information systems. We measure the flow intensity at multiple checkpoints inside the system and apply system identification methods to model transaction flow dynamics between these measurements. With the learned analytical models, a model-based fault detection and isolation method is applied to track the flow dynamics in real time for fault detection. We also propose an algorithm to automatically search and validate the dynamic relationship between randomly selected monitoring points. Our algorithm enables systems to have self-cognition capability for system management. Our approach is tested in a real system with a list of injected faults. Experimental results demonstrate the effectiveness of our approach and algorithms.

Index Terms—Fault detection, information systems, system management, regression model, model-based FDI, dynamic relationship, model validation, flow intensity and dynamics.

1 Introduction

In the last 10 years, we have witnessed the great success of Internet services. Numerous Internet services such as Amazon, ebay, and Google are dramatically changing traditional business models. With the prevalence of such Internet services, there are unprecedented needs to ensure their operational reliability and availability. Minutes of service downtime could lead to severe revenue loss and users' dissatisfaction [1]. Meanwhile, the information system of an Internet service is usually a large, dynamic, and distributed system and could consist of thousands of components. A single fault in its components could make the whole service unavailable [2]. Software faults are notoriously difficult to eradicate, so that it is unrealistic to expect that a system of high complexity is fault-free. Studies have shown that the time taken to detect and isolate faults contributes 75 percent of the failure recovery time [3]. Therefore, it has become a major challenge for system management to detect and isolate faults effectively in such a large and complex system.

A large amount of monitoring data can be collected from system components for fault analysis. Software log files, system audit events, and network traffic statistics are typical examples of such monitoring data. If we regard the operational system as a dynamic system, this data is the observable of its internal states regarding system "health." Given the distributed nature of complex information system, evidence of fault occurrence is often scattered among the monitoring data. A critical challenge is how to correlate this data effectively across distributed systems and observation time for fault detection and isolation. Dozens of vendors have provided advanced monitoring and management tools for system administrators to interpret the monitoring data. IBM's Tivoli [4], HP's OpenView [5], and EMC's InCharge [6] are the leading products in this growing market of system management software. Most current tools support some form of data preprocessing and enable users to view the data with visualization functions. These tools are useful because it is impossible for system administrators to manually scan a large amount of raw monitoring data. However, most tools only employ simple rule-based correlation with little embedded intelligence for reasoning. For example, the tools are set to generate alerts based on certain threshold violations. Rule-based systems are also inherently stateless and do not handle dynamic data analysis well. The lack of intelligence mainly results from the difficulty in characterizing dynamic behavior of complex systems. We believe that much of this knowledge is inherently system-dependent, i.e., it is hard to generalize such knowledge across systems with different architecture and functionality. Therefore, for complex systems, it is very desirable to develop system self-cognition capability.

Internet services receive a huge number of transaction requests from users every day, and these requests flow through the set of components according to specific application software logic. With such a large volume of user visits, we believe that it is not realistic to monitor and analyze each individual user request. Instead, we should consider the mass characteristics of user request flows as we do with the concept of pressure and energy in physics. In this paper, we propose novel concepts, *flow intensity* and *flow dynamics*, to capture the dynamics of mass user request flows in Internet services. Further, we propose a novel approach to model and track transaction flow dynamics for fault detection. We monitor and calculate the flow intensity

• *The authors are with NEC Laboratories America Inc., 4 Independence Way, Princeton, NJ 08540. E-mail: {gfj, haifeng, kenji}@nec-labs.com.*

Manuscript received 14 July 2005; revised 27 Jan. 2006; accepted 24 Apr. 2006; published online 2 Nov. 2006.
For information on obtaining reprints of this article, please send e-mail to: tdsc@computer.org, and reference IEEECS Log Number TDSC-0105-0705.

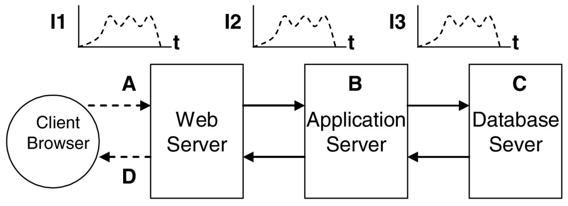


Fig. 1. Three-tier architecture.

(e.g., the number of requests per sampling time) at multiple checkpoints inside the system. These flow intensity measurements are regarded as input and output of various system segments and they are used to model the flow dynamics with system identification methods. The resulted analytical models reflect the flow dynamics that the normal system should bear. For example, a surge of HTTP requests to the front Web server usually leads to a similar surge of SQL requests to the back-end database server. If such relationships hold all the time, we can regard them as invariants of the dynamic system. If a fault occurs inside the system, the flow dynamics are likely to be affected and some of the invariant relationships are likely to be violated. Therefore, we could detect such a fault in real time by tracking the change of invariants.

This paper presents a new framework for fault detection and isolation in complex information systems. In the following sections, we will first introduce basic concepts of our approach and then discuss each individual component of this framework. This paper makes the following contributions:

- We propose a novel concept, named *transaction flow dynamics*, to represent the dynamic relationship between the flow intensity measured at multiple points across distributed systems. System identification methods are applied to automatically learn the regression models that characterize the flow dynamics in a specific system.
- With the learned models, we propose a model-based Fault Detection and Isolation (FDI) method to track transaction flow dynamics in real time for fault detection. Though this approach has been well studied in control theory [7], [8], to the best of our knowledge, we have not seen its application for fault detection in complex information systems.
- We propose an algorithm to automatically search and validate the relationship between flow intensity measurements at multiple points. Sequential testing is applied to derive the confidence on whether such a dynamic relationship holds for two randomly selected monitoring points. This confidence score is further used to evaluate the credibility of detection results. Our algorithm enables systems to have a certain level of self-cognition capability for system management.
- Our fault detection approach is tested in a real system with a list of injected faults. The experimental results demonstrate the effectiveness of our approach. Our model search algorithm is also tested and proved to be able to identify many of such dynamic relationships. Our experiments also reveal an important fact that such invariant relationships widely exist in distributed transaction systems.

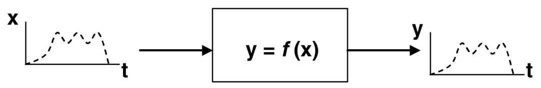


Fig. 2. Flow intensity and flow dynamics.

2 Transaction Flow Intensity and Dynamics

Many Internet services employ multitier architectures to integrate their components. A typical three-tier architecture is illustrated in Fig. 1, which includes a Web server, an application server (middleware), and a database server. The front Web server acts as an interface to present data to the client's browser. The middle application server supports specific business logic for various applications, which generally represents the bulk of an application. The back-end database server is for persistent data storage. Each tier can be built from a number of software packages providing similar functionality. For example, Apache or IIS for the Web server, WebLogic, or WebSphere for the application server, and Oracle or DB2 for the database server.

Many Internet services receive millions of user visits every day. Some recent statistics have shown that eBay received 65.9 million user visits in the month of March 2005. Amazon sold 3.6 million items or 41 items per second in the single day of 12 December 2005 [9]. With such a large number of user visits, we believe that it is not realistic to track each user request to verify its correctness. Instead, we should analyze the mass characteristics of user request flows as we do with the concepts of pressure and energy in physics, which is our main motivation for this work.

User requests traverse the set of components according to specific application logic. If we regard the control flow graph of application software as a complex pipe network, the mass of user requests flows through various software paths as fluid flows through that pipe network. Many internal monitoring data collected at various points reacts to the volume of user requests accordingly. We propose a concept, named *flow intensity*, to measure the intensity with which the internal monitoring data reacts to the volume of user requests. In Fig. 1, A, B, and C are such checkpoints in the three-tier system and $I1$, $I2$, and $I3$ are the flow intensities measured at these points, respectively. If we consider a component as a black box, the flow intensity measured at the input and output of this box could well reflect the flow dynamics that this component should bear. An example is given in Fig. 2 to illustrate the concept of transaction flow dynamics. The flow dynamics $y = f(x)$ is determined by the internal structure or constraint of the box, where the input x and the output y are both time series. If a fault's occurrence changes its structure or constraint, its flow dynamics is likely to be affected and the dynamic relationship (the equation) $y = f(x)$ could be violated. Therefore, we could detect such a fault by tracking whether such a relationship continues to hold. From this perspective, endless user requests act like "probing and testing" the system all the time.

In practice, it is usually difficult to count the exact number of user requests at many monitoring points inside the system. Instead, we can use other common and lightweight measurements to represent the flow intensity. For the three-tier system shown in Fig. 1, we give a list of measurement examples that can be used to represent the flow intensity:

- Web server: Based on the Web access log, we can extract the number of users, the number of HTTP requests received and completed, the number of sessions, etc.
- Application server: Based on the data collected from lightweight monitoring mechanisms such as JMX [10], we can extract the number of live threads, the amount of heap memory usage, the number of in-use database connections, the number of processing EJBs, etc.
- Database server: Some monitoring tools enable us to count the number of SQL queries.
- Networking: Many tools enable us to collect statistical data about network traffic and activity.
- Operating System: For all these servers, OS monitoring tools allow us to collect data about CPU, memory and disk usage, etc.

Note that multiple flow intensity measurements can be derived from one single monitoring point in the system. For example, we can calculate several flow intensity measurements from Web server access logs. As mentioned earlier, dozens of vendors offer advanced monitoring and management tools to collect a lot of monitoring data from distributed systems. For example, AdventNet's ManageEngine software [11] monitors and collects hundreds of such measurements for a typical three-tier system. Our framework could import measurements from these monitoring tools. In this paper, we focus on how to interpret such a large amount of monitoring data for fault analysis.

3 Modeling Flow Dynamics

With the flow intensity measured at multiple points in distributed systems, we need to consider how to model the flow dynamics between these monitoring points. For mechanical or electronic systems, we can apply first principles such as physical laws to derive analytical models. This is usually not feasible for complex systems, where a model has to be learned based on empirical data. Since an Internet service is an operational system and typically has a large number of user visits, we should have sufficient data to learn such a model. In this paper, we use AutoRegressive models with eXogenous inputs (ARX) [12] to learn the relationship between flow intensity measurements. Note that other data relationship models such as the Gaussian mixture model [13] can also be used to characterize the flow dynamics in our framework. Without loss of generality, here we use the popular ARX model as an example to illustrate our concept.

At time t, we denote the flow intensity measured at the input and output of a component by $x(t)$ and $y(t)$, respectively. The ARX model describes the following dynamic relationship between the input and output:

$$\begin{aligned} y(t) + a_1 y(t-1) + \cdots + a_n y(t-n) \\ = b_0 x(t-k) + \cdots + b_m x(t-k-m), \end{aligned} \tag{1}$$

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. Let's denote

$$\theta = [a_1, \cdots, a_n, b_0, \cdots, b_m]^T, \tag{2}$$

$$\begin{aligned} \varphi(t) = [-y(t-1), \, \ldots, -y(t-n), \\ x(t-k), \, \ldots, x(t-k-m)]^T. \end{aligned} \tag{3}$$

Then, (1) can be rewritten as:

$$y(t) = \varphi(t)^T \theta. \tag{4}$$

Assuming that we have observed the inputs and outputs (i.e., the flow intensity) over a time interval $1 \leq t \leq N$, let's denote this observation by:

$$O_N = \{x(1), y(1), \, \ldots, x(N), y(N)\}. \tag{5}$$

For a given θ, we can use the observed inputs $x(t)$ to calculate the simulated outputs $\hat{y}(t|\theta)$ according to (1). Thus, we can compare the simulated outputs with the real observed outputs and define the estimation error by:

$$\begin{aligned} E_N(\theta, O_N) &= \frac{1}{N}\sum_{t=1}^{N}(y(t) - \hat{y}(t|\theta))^2 \\ &= \frac{1}{N}\sum_{t=1}^{N}(y(t) - \varphi(t)^T\theta)^2. \end{aligned} \tag{6}$$

The Least Squares Method (LSM) can find the following $\hat{\theta}$ that minimizes the estimation error $E_N(\theta, O_N)$:

$$\hat{\theta}_N = \left[\sum_{t=1}^{N}\varphi(t)\varphi(t)^T\right]^{-1}\sum_{t=1}^{N}\varphi(t)y(t). \tag{7}$$

Note that there are recursive algorithms to compute the $\hat{\theta}$ and the ARX model can also be used to model the relationship between multiple inputs and multiple outputs [12], i.e., we can have multiple flow intensity measurements as the inputs and/or outputs in (1). For simplicity, to introduce our concept, we only model and analyze the relationship between a single input and a single output in this paper.

There are several criteria to evaluate how well the learned model fits the real observation. In this paper, we use the following equation to calculate a normalized fitness score for model validation:

$$F(\theta) = \left[1 - \sqrt{\frac{\sum_{t=1}^{N}|y(t) - \hat{y}(t|\theta)|^2}{\sum_{t=1}^{N}|y(t) - \bar{y}|^2}}\right] \cdot 100, \tag{8}$$

where $\bar{y}$ is the mean of the real output $y(t)$. A higher fitness score indicates that the model fits the observed data better and its upper bound is 100. As the order of the model structure $[n, m]$ increases, the fitness score monotonically decreases and we could learn models that overfit the data. The increasing flexibility of the model structure eventually enables the model to fit noise well (i.e., overfit); however, noise changes randomly over time and does not reflect the real flow dynamics. Several criteria were proposed to indicate the goodness of a model such as Akaike's Information-theoretic Criterion (AIC) [14] and Rissanen's Minimum Description Length (MDL) [15]. Mathematical software tools such as Matlab have standard functions that implement these criteria. Given the observation of two flow intensities, we can always use (7) to learn a model even if this model does not reflect their relationship at all. Therefore, only a model with a high fitness score is really meaningful in characterizing data relationship. Therefore, we can set a range of the order $[n, m, k]$ rather than a fixed number to learn a list of model

```
Algorithm 3.1
Input: $x(t)$, $y(t)$ and a list of orders $\{[n,m,k]_i\}$
Output: model parameter $\hat{\theta}$

for each $[n,m,k]_i$,
    compute $\hat{\theta}_i$ with Equation (7);
    compute $F(\hat{\theta}_i)$ with Equation (8).
$\hat{\theta} = \arg max_{\hat{\theta}_i} F(\hat{\theta}_i)$ or AIC($\hat{\theta}_i$) or MDL($\hat{\theta}_i$).
return $\hat{\theta}$.
```

Fig. 3. Flow dynamics modeling algorithm.

candidates and then a right model can be selected from them according to these criteria. For the completeness of this paper, we summarize the above modeling steps with the learning algorithm from Fig. 3.

Now, the question is whether we can use such a linear regression model to capture the dynamic relationship between flow intensity measurements in real systems. Intuitively, some of the measurements listed in Section 2 may have such relationships. For example, a surge of HTTP requests to the Web server usually leads to a similar surge of SQL requests to the database server because some percentage of HTTP requests have to access the database. Many flow intensity measurements react to the same workload accordingly, so they should have linear relationships between them. At this point, it is useful to analyze some real monitoring data from our testbed system. Our testbed system is a realistic three-tier e-commerce system with emulated user requests and scenarios as workloads. We will introduce the details of our testbed system in Section 7. Here, we choose three monitoring points to calculate flow intensity: Apache Web server's HTTP access log, JBoss [16] application server's CPU usage, and MySQL database server's SQL queries. Denote the intensity measured from these three points by I_{web}, I_{cpu}, and I_{sql}, respectively. Since the Apache log includes the information about the time taken to complete each HTTP request, we calculate two intensity measurements: I^r_{web} for the flow intensity that enters the Apache server (the point A in Fig. 1) and I^l_{web} for the flow intensity that leaves the Apache server (the point D in Fig. 1).

Fig. 4 shows the four intensity measurements during a one and a half hour period. Our sampling time is 10 seconds and the Y axis shows the average intensity (per second) during each sampling time. Note that the best sampling time is dependent on the dynamics of modeled transaction flows. For example, if the flow dynamics varies quickly, its sampling rate has to be high so as to capture its dynamics. From the above figures, it is straightforward to find that the curves of I^r_{web}, I^l_{web}, and I_{sql} are very similar, which implies that there are strong linear relationships between these measurements. The original curve of I_{cpu} (the thin line) shows some high frequency cycles, which result from JVM's periodical garbage collection operations. We use a low-pass filter to filter out the signal with frequency higher than 0.05 rad/s, which results in the filtered curve of I_{cpu} (the thick line). It is clear that the filtered curve of I_{cpu} also has a shape similar to those of other measurements. In the following sections, we will use this filtered I_{cpu} in analysis.

For convenience, here we choose I^r_{web} as the input to model its relationship with I_{cpu}, I_{sql}, and I^l_{web}, respectively. Studies have shown that users are usually not satisfied with Internet service performance if the latency of their requests

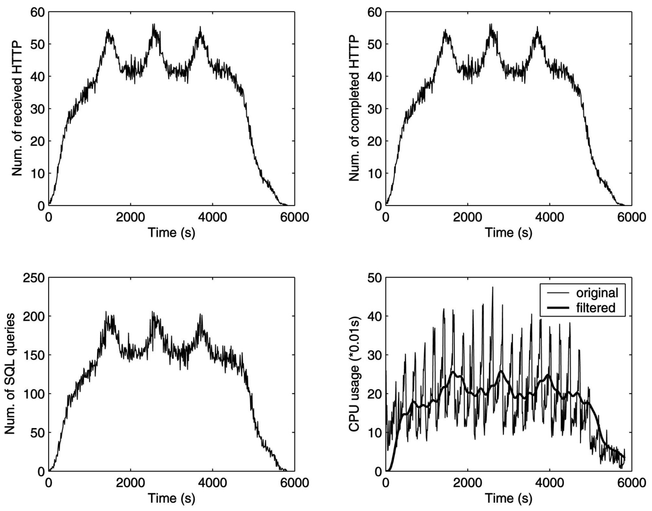


Fig. 4. Intensity measurement at multiple points.

is more than 10 seconds. Our sampling time is 10 seconds because we don't want to introduce much overhead in monitoring. Therefore, we believe that the output flow intensity will not be delayed much than the input flow intensity and the range of the model orders should be very narrow. With Algorithm 3.1 (Fig. 3), we learn the following models:

$$I_{cpu}(t) = 1.968 I_{cpu}(t-1) - 0.972 I_{cpu}(t-2) + 0.002 I^r_{web}(t), \tag{9}$$

$$I_{sql}(t) = 0.2099 I_{sql}(t-1) + 0.2931 I_{sql}(t-2) + 1.791 I^r_{web}(t), \tag{10}$$

$$I^l_{web}(t) = I^r_{web}(t). \tag{11}$$

We verified that these models fit the real observed data well under many different workloads and user scenarios. More details will be discussed in Section 7. For some models, it is easy to understand their physical meanings. For example, (11) implies that all user requests were completed in less than 10 seconds (the sampling time). However, for some models such as (9), the physical meanings of their parameters are not obvious. If the structure of a model is known and given, the learned parameters usually represent some physical meanings. For a complex system, we seldom have such knowledge about model structures. Though system identification approaches can be applied to learn a model that fits the observed data well, it is usually difficult to interpret the physical meanings of its parameters. For example, multiple models could fit the observed data well but there is only one real structure of a specific system. However, as long as such a learned model can always fit the observed data well in a dynamic system, we can use such a model/relationship as an oracle in fault detection and we don't have to interpret the real structure of systems.

4 Model-Based Fault Detection

If these models/relationships continue to hold all the time under different user scenarios and workloads, we can regard them as "invariants" of the monitored system. One

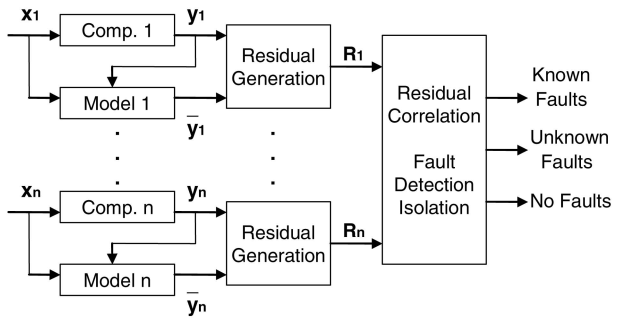


Fig. 5. Structure of the model-based FDI.

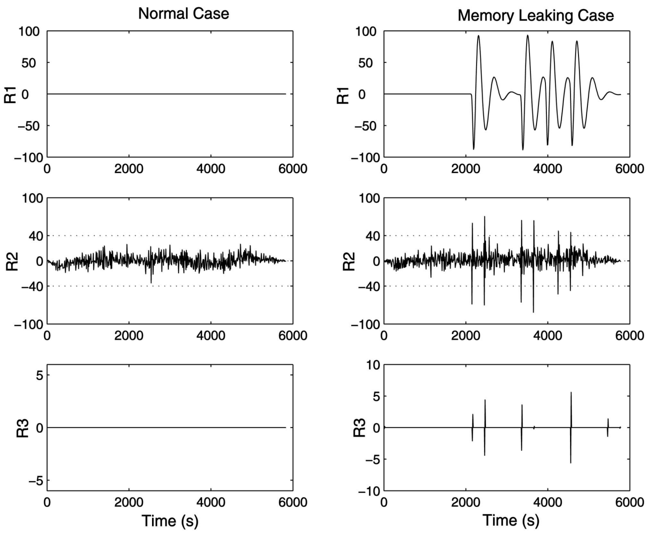


Fig. 6. Comparison of residuals.

concern is whether such invariants are still valid after user behavior changes. For example, the distribution of HTTP requests to various pages of a Web server is significantly changed. This change could affect the relationship between I^r_{web} and I_{sql} because some HTTP requests could access the back-end database server more frequently than others. Conversely, this change will not affect the relationship between I^r_{web} and I^l_{web} in a normal situation. In general, we believe that some of these relationships could be sensitive to specific environmental changes. In Section 6, we will introduce an automated model validation algorithm to dynamically track whether such a model/relationship continues to hold, and we will derive a confidence score to evaluate its "validity." Those unstable models are discarded and not used in fault detection and isolation.

Meanwhile, we believe that the mass characteristics of user requests will not change much. Internet services receive a large number of user requests during each time period and most of these requests actually focus on several major functions. An important fact is that each user's behavior is usually independent due to the nature of Internet services. With a large number of independent users, according to the Central Limit Theorem in statistics [17], the sampling distribution of user scenarios always follows a normal distribution with narrow variance. Therefore, we believe that the randomness of individual user behavior will not change the mass characteristics much and many learned models/relationships of flow intensity will continue to hold. However, some external factors could lead to emergent user behavior. For example, on sale advertisements could suddenly cause collective user behavior, which may violate some of the invariants. As mentioned above, in Section 6, we have an algorithm to dynamically track the model validity over time. In addition, many invariant relationships do not change no matter how user behavior varies. For example, if one specific HTTP request always leads to two related SQL queries as written in the application logic, this relationship should always hold no matter how user behavior varies.

As briefly discussed in Section 2, a fault's occurrence inside the monitored system could break some of these invariants. Therefore, we can detect such a fault by tracking whether the real outputs stay on the trajectory expected by the models. The model-based FDI approach is based on the use of analytical redundancy, i.e., comparing the real system with a mathematical model. This approach has been well analyzed in control theory and engineering, but, to the best of our knowledge, it has not been applied to complex information systems. One probable reason is that it is not easy to obtain analytical models in a complex information system. Fig. 5 shows the basic structure of the model-based FDI approach. Here, we use the general term "component" to represent a segment of the monitored system such as the segment between the points A and C in Fig. 1. Model k is the learned analytical model that describes the flow dynamics for the $k\text{th}$ component. For example, (9), (10), and (11) are such analytical models. x_k and y_k are the flow intensities measured at the input and output of the real component k. $\overline{y}_k$ is the simulated output of the model k when fed with the real x_k and y_k, i.e., $\overline{y}_k = f_k(x_k, y_k)$, where f_k represents the model k. Note that the real outputs y_k observed at the earlier time steps are needed to compute the current simulated output $\overline{y}_k$. For example, in (9), at time t, we need $I_{cpu}(t-1)$ and $I_{cpu}(t-2)$ to compute the current output $\overline{I}_{cpu}(t)$. Instead of using the simulated output $\overline{I}_{cpu}(t-1)$ and $\overline{I}_{cpu}(t-2)$, we use the real output observed at the earlier time steps to compute the current simulated output. Otherwise, the estimation error will be accumulated over the time. Thus, we can generate the $k\text{th}$ residual by

$$R_k(t) = y_k(t) - \overline{y}_k(t). \tag{12}$$

If we model n of such dynamic relationships among flow intensity measurements, we can generate n such residuals. In a normal situation, these residuals can be regarded as the noise resulting from modeling and they are usually small. If some fault occurs inside the system, some of these n relationships could be affected and their residuals become significantly larger. Therefore, by tracking these residuals, we could detect such a fault. Many faults could cause service failure or performance deterioration in a complex system, which include various software bugs, hardware problems, network faults, and operator mistakes. Due to the diversity of faults, it is impossible to use one general pattern to detect all these faults. In fact, various faults affect a system in different ways and they can only be detected using different patterns. Since we have n models rather than one for fault detection, we believe that our approach could detect a wide class of faults by tracking these n residuals. Note that these n models characterize the normal behavior of the monitored system from various perspectives, as exampled by (9), (10), and (11). In Section 6, we will discuss how to automatically discover as many such relationships as possible based on collected intensity measurements.

```
Algorithm 4.1
Input: $u_k(t)$, $y_k(t)$ and a model $f_k$
Output: abnormal residual alert
for each time step $t$,
    compute the simulated output $\overline{y}_k(t)$
        using the given model $f_k$;
    compute the residual $R_k(t) = y_k(t) - \overline{y}_k(t)$;
    if $|R_k(t)| > \tau$,
        then generate an alert;
```

Fig. 7. Online tracking and detection algorithm.

As an example, Fig. 6 shows the residuals generated from a normal case (the left figures) as well as a memory leaking case (the right figures). The memory leaking fault occurred in the application software running on the JBoss middleware. Equations (9), (10), and (11) are the analytical models used to generate the residuals R_1, R_2, and R_3, respectively.

$$R_1(t) = I_{cpu}(t) - \overline{I}_{cpu}(t), \tag{13}$$

$$R_2(t) = I_{sql}(t) - \overline{I}_{sql}(t), \tag{14}$$

$$R_3(t) = I^l_{web}(t) - \overline{I}^l_{web}(t). \tag{15}$$

$I_{cpu}(t)$, $I_{sql}(t)$, and $I^l_{web}(t)$ are the real intensity measurements while $\overline{I}_{cpu}(t)$, $\overline{I}_{sql}(t)$, and $\overline{I}^l_{web}(t)$ are the simulated outputs of the analytical models fed with the real input $I^r_{web}(t)$. From these figures, it is straightforward to see that this fault causes some peaks in all three residuals, which are significantly larger than the normal noise shown in the left figures. Meanwhile, all of these three residuals start to indicate the existence of the fault at the same time (some time around 2,000 seconds). In fact, this is the time that the memory leaking fault eventually triggers intensive garbage collection operations of JVM. Since this intensive operation is not caused by the intensity of the workload, the normal flow dynamics is affected and the real outputs stay out of the tracks expected by the analytical models. Note that the memory leaking fault periodically delays but does not block user requests and every request is still completed. Therefore, the number of delayed user requests would decrease the flow intensity during one time unit but increase the flow intensity at the following time unit. Consequently, a positive peak is always coupled with a negative peak in the figures.

More examples of fault detection will be discussed in Section 8. In general, we need a threshold to determine the abnormality of a residual. Theoretically, this threshold can only be optimized if the distribution of residuals resulting from various faults is known. In practice, since the residuals are tracked over time, we can use the statistics of their past values to derive a dynamic threshold. For example, we can select a threshold $\tau = 1.1 \cdot \arg_{\hat{R}}\{prob(|R(t)| < \hat{R}) = 0.995\}$, i.e., choose a value $\hat{R}$ that is larger than 99.5 percent of the observed residuals (after a long time period t) and the selected threshold is 1.1 times $\hat{R}$. Since faulty situations are very rare in operations and normal situations are usually dominant, most observed residuals are normal noise. An alert is generated only if a residual is much larger than the 99.5 percent of its past values. We complete this section with the online tracking and detection algorithm from Fig. 7.

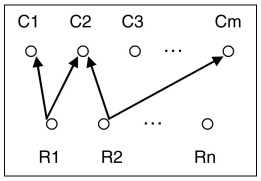


Fig. 8. Residuals and their monitoring components.

5 Residual Correlation and Fault Isolation

As shown in Fig. 5, n models of flow dynamics generate n residuals. While some faults may cause several residuals to be abnormal, some other faults may leave little evidence of their existence in residuals. Therefore, whether a specific fault can be detected is dependent on its impact on these residuals. If we have many such residuals, straightforwardly we would expect to detect a wide class of faults in the system. A critical challenge is how to correlate these n residuals for fault isolation.

Since each of our models describes the flow dynamics for a specific segment of the system, each of these residuals could be used to indicate whether a specific part of the monitored system is healthy. For example, the residual generated from (9) could indicate the occurrence of faults in the Web server and/or application server but not the database server. Some of these models may have overlaps with regard to their monitoring components. Relationships between residuals and their monitoring components are illustrated in Fig. 8. A pointer means that the pointed component is monitored by the residual at the starting point. Based on the causal relationship illustrated in Fig. 8, we can develop a residual correlation matrix as illustrated in Fig. 9. The columns represent the components of a monitored system while the rows represent residuals. The element of the matrix $M_{ij} = 1$ means that the component C_j is monitored by the residual R_i. Otherwise, $M_{ij} = 0$.

At time t, we use a binary value $r_i(t)$ to classify the abnormality of the residual $R_i(t)$. $r_i(t) = 1$ means that the residual $|R_i(t)| > \tau$. Conversely, $r_i(t) = 0$ means that $|R_i(t)| \leq \tau$. Thus, with all n residuals we can define an observation vector $O(t) = [r_1(t), r_2(t), \cdots, r_n(t)]^T$. Here, we also use C_j to represent the jth column of the correlation matrix. In normal situations, we should have $\|O(t)\| = 0$. If $\|O(t)\| \geq 1$, we can use the following Jaccard coefficient [18] to locate the faulty component C_f:

	C1	C2	...	Cm
R1	1	1	...	0
R2	0	1	...	1
...	...	...	...	...
Rn	0	0	...	0

Fig. 9. Residual correlation matrix.

$$C_f = \arg max_j V_j, \quad where \;\; V_j = \frac{\sum_{i=1}^{n} r_i(t) \cap C_{ij}}{\sum_{i=1}^{n} r_i(t) \cup C_{ij}}. \qquad (16)$$

Equation (16) identifies the component, which is maximally correlated with the residual observation, as the faulty component. The underlying assumption is that the faulty component would cause many of its related residuals to be abnormal. Rather than using (16) to locate one most suspicious component, we can also identify a set of suspicious components by ranking their values of V_j.

The granularity of fault isolation depends on the monitoring data we can collect from a system and the invariants we can extract from the data. In general, with more models (i.e., bigger n), we should be able to isolate faulty components at a finer granularity. Meanwhile, even a coarse level isolation (e.g., at the machine level) could still be useful for troubleshooting in complex information systems. At first, system administrators could consult the isolation result to further narrow down suspicious components using other fine granularity diagnosis tools. Second, many complex information systems have redundant architecture. We can use a load balancer to switch user requests to other peers and reboot the suspicious components to recover, without knowing the root cause of faults.

Besides the location of faults, we also want to know the type of faults in troubleshooting. In fact, a residual provides much more information than its binary representation. For example, in the memory leaking case shown in Fig. 6, positive peaks are always coupled with negative peaks. The mean of the residuals has to be close to zero because every request is eventually completed. Conversely, as we will see in Section 8, a logic fault of software usually stops some user requests from being completed. Therefore, we will see a thoroughly different shape of the residuals and the residual mean is also biased from zero. We believe that such information can be used to distinguish the type of faults as illustrated in [19], [20].

Based on cause-effect analysis and system administrator's experience, we can develop the knowledge between a list of faults and their symptoms on these residuals. A similar correlation matrix could be formulated to describe the relationships between residuals (the rows) and known fault types (the columns). Therefore, we need to input administrators' expert knowledge for this process, especially if a specific fault or symptom has been diagnosed before. In some complex systems, it may be difficult to get the dependency knowledge shown in Fig. 8. In that case, we can formulate a correlation matrix between residuals (the rows) and their related measurements (the columns), where each residual is always related to two measurements and this dependency relationship is known. Further, we can use the similar approach to locate the most suspicious measurement first and then follow the broken invariants from that measurement to narrow down the faulty components. Note that many other evidential reasoning approaches can also be used for fault isolation such as kernel classifier [21], Bayesian network, fuzzy logic, and Dempster-Shafer theory [22]. In general, these approaches need more statistical information than that used in the correlation matrix. We will investigate these approaches in our future work.

6 Automated Model Search and Validation

In Section 3, we analyzed how to automatically build a model between two flow intensity measurements. In a complex system, we may collect a large number of flow intensity measurements but obviously not all pairs would have such linear relationships. Meantime, due to system dynamics and uncertainties, some learned models may not be robust along time. The question is how to determine whether there are such invariant relationships among measurements. As discussed earlier, we want to use as many invariants as possible in fault detection and isolation. In general, with more invariants, we could detect a wider class of faults, and isolate and distinguish faults at a finer granularity. In practice, we may build some of these relationships based on prior system knowledge. However, we believe that this knowledge is very limited and system dependent due to various architecture and functionality of complex information systems. To this end, we propose an algorithm to automate the process of model search and validation.

Assume that we have m flow intensity measurements denoted by I_i, $1 \leq i \leq m$. Without loss of generality, here, we just check the relationship between each pair of measurements in this section. Since we have little knowledge about the relationship among these measurements from a specific system, we try any combination of two measurements to construct a model first and then continue to validate this model with new incoming observations, i.e., we first use brute-force search to discover models and then sequentially test the validity of these models in operation. The fitness score $F_i(\theta)$ given by (8) is used to evaluate how well a learned model matches the data observed during the ith time window. We denote the length of this window by l, i.e., the window includes l sampling points. We select a threshold $\widetilde{F}$ and use the following piecewise function to determine whether a model fits the data or not.

$$f(F_i(\theta)) = \begin{cases} 1 & \text{if } F_i(\theta) > \widetilde{F}, \\ 0 & \text{if } F_i(\theta) \leq \widetilde{F}. \end{cases} \qquad (17)$$

After receiving the monitoring data for k of such windows, i.e., total $k \cdot l$ sampling points, we can calculate a confidence score with the following equation:

$$\begin{aligned} p_k(\theta) = prob(F_t(\theta) > \widetilde{F}) &= \frac{\sum_{i=1}^{k} f(F_i(\theta))}{k} \\ &= \frac{p_{k-1}(\theta) \cdot (k-1) + f(F_k(\theta))}{k}. \end{aligned} \qquad (18)$$

We can also use average fitness score to derive the above confidence score, i.e., in (17), if $F_i(\theta) > \widetilde{F}$, let $f(F_i(\theta)) = F_i(\theta)/100$ and otherwise $f(F_i(\theta)) = 0$. Note that the upper bound of fitness scores is 100. Without loss of generality, we use (17) in the following analysis. As shown in the second part of the above equation, the value of $p_k(\theta)$ will be affected little by the current fitness score once k becomes large. Therefore, we can use the latest L windows (i.e., from $k - L$ to k) rather than the total k windows to compute the current $p_k(\theta)$. Denote the valid set of models at time $t = k \cdot l$ by M_k, i.e., $M_k = \{\theta | p_k(\theta) > P\}$. P is the confidence threshold we choose to determine whether a model is valid. The automated model search and validation algorithm is shown in Fig. 10. Algorithm 6.1 in Fig. 10 first starts to build a model for any two measurements and then incrementally validates these models with new observations. After a time period ($K \cdot l$ sampling points and K is a selected number), if the confidence score of a model is less than the selected threshold P, we consider this model invalid and stop validating this model as invariant. Since the occurrence of faults could deteriorate $p_k(\theta)$ temporally, we need several

```
Algorithm 6.1

Input: $I_i(t)$, $1 \le i \le m$
Output: $M_k$ and $p_k(\theta)$ for each time window $k$

at time $t = l$ (i.e., $k = 1$),
    for each $I_i$ and $I_j$, $1 \le i, j \le m$, $i \ne j$
        learn a model $\theta_{ij}$ using Algorithm 3.1;
        compute $F_1(\theta_{ij})$ with Equation (8).
        set $M_1 = M_1 \cup \{\theta_{ij}\}$ and $p_1(\theta_{ij}) = f(F_1(\theta_{ij}))$.

for each time $t = k \cdot l$, $k > 1$,
    for each $\theta_{ij} \in M_k$,
        compute the $F_k(\theta_{ij})$ with Equation (8) using
            $I_i(t)$ and $I_j(t)$, $(k-1) \cdot l + 1 \le t \le k \cdot l$;
        update $p_k(\theta_{ij})$ with Equation (18);
        if $p_k(\theta_{ij}) \le P$ and $k \ge K$,
            then remove $\theta_{ij}$ from the $M_k$.
    output $M_k$ and $p_k(\theta)$.
    $k = k + 1$.
```

Fig. 10. Model search and validation algorithm.

time windows of monitoring data to make sure that a model is invalid. Note that in an operational environment like Internet services, the normal situation is dominant and faulty situations are very rare. Therefore, we can determine whether a model is invalid with a small K.

For model search and validation purposes, another high threshold $\hat{P}$ can be chosen to determine whether we can stop validating a good model, i.e., we already have enough confidence to conclude that a model is valid. Thus, our sequential testing could have the following stopping rules: at time $t = k \cdot l$, if $p_k(\theta) \ge \hat{P}$, accept and stop validating this model; if $p_k(\theta) \le P$, reject and stop validating this model; if $P < p_k(\theta) < \hat{P}$, continue to validate this model. In the last case, we don't have enough confidence to decide whether this model should be accepted up to this time point so that we continue to validate this model with new incoming observation. In Algorithm 6.1 (Fig. 10), we choose not to stop validating good models but keep updating $p_k(\theta)$ for two reasons: At first, we want to keep tracking the robustness of valid models. If evolution occurs, we may want to relearn this relationship with Algorithm 3.1 (Fig. 3). Second, we use the updated $p_k(\theta)$ as a confidence score to evaluate the credibility of the residual generated from this model. For example, with a higher $p_k(\theta)$, we should be more confident about the detection result generated from this model, assuming that the residual threshold is correctly selected. Conversely, if the $p_k(\theta)$ of a model is relatively low, the modeled relationship may not be strong and we should be conservative about the correctness of its residual in detection. As discussed earlier, in order to keep $p_k(\theta)$ up to date, we can use the latest L windows (instead of the total k windows) to compute $p_k(\theta)$.

We can use a large amount of data for training because the monitoring data from $24 * 365$ operational environments is sufficient. As long as normal behavior is dominant in the collected training data, our approach is effective even if the training data includes faulty situations. This condition can be easily satisfied by using a large amount of training data, because system behavior is usually dominated by normal behavior in reasonably well managed Internet services. Our approach models dominant long-run relationships between measurements with

Residual	Observation	Confidence
R1	$r_1(t)$	$p_k(\theta_1)$
R2	$r_2(t)$	$p_k(\theta_2)$
...	...	...
Rn	$r_n(t)$	$p_k(\theta_n)$

Fig. 11. Confidence score of residuals.

little attention to rare outliers in the training data. This can be explained with (6) in Section 3. We learn a model by minimizing the estimation error $E_N(\theta, O_N)$. According to (6), with a large N, several "peaks" of the error $(y(t) - \hat{y}(t|\theta))^2$ (caused by faults) will change the total value of $E_N(\theta, O_N)$ little, so the learned model will still capture the mass characteristics of their long-run relationship. Meanwhile, a large amount of training data is also necessary for a model to capture various flow dynamics that could happen in monitored systems.

As mentioned above, Algorithm 6.1 (Fig. 10) keeps updating the confidence score $p_k(\theta)$ for each valid model θ. Since residuals are generated from these models as shown in (12), we can use this confidence score to evaluate the credibility of residuals. For detection, a residual generated from a high quality model (with high $p_k(\theta)$) is clearly more credible than that from a low quality model (with low $p_k(\theta)$). As introduced in Section 5, at time t, we use a binary value $r_i(t)$ to represent the abnormality of the residual $R_i(t)$. $r_i(t) = 1$ means that the residual $|R_i(t)| > \tau$. Conversely, $r_i(t) = 0$ means that $|R_i(t)| \le \tau$. Thus, with all n residuals, we can define an observation vector $O(t) = [r_1(t), r_2(t), \ldots, r_n(t)]^T$. We build a table as shown in Fig. 11 to illustrate the relationship between $r_i(t)$ and $p_k(\theta_i)$. Note that $p_k(\theta_i)$ is updated once for each window with l sampling points. Here, we assume $(k-1) \cdot l + 1 < t \le k \cdot l$ in this table.

In Section 5, as long as $\|O(t)\| \ge 1$, we conclude that there are faults inside the system. This decision rule could cause false positives if some models with relatively low $p_k(\theta)$ are used in detection. These models could generate large residuals under some unexpected situations because the model itself may not capture the flow dynamics well enough for these situations, i.e., the large residual results from the inaccurate model itself rather than any real faults. Meanwhile, as discussed earlier, we also want to have as many models as possible for fault analysis and these unstable models could still be useful at most time. To this end, we propose using the following weighted score $s(t)$ to determine whether a system is faulty at time t:

$$s(t) = \frac{\sum_{i=1}^{n} r_i(t) \cdot p_k(\theta_i)}{n}. \tag{19}$$

A threshold S should be chosen to determine whether $s(t)$ indicates the occurrence of faults. As long as $s(t) > S$, we believe that the system is faulty. Note that $r_i(t)$ is either one or zero. If a model with high $p_k(\theta_j)$ generates an abnormal residual (i.e., $r_j(t) = 1$), the score $s(t)$ will be high. This is reasonable because we have high confidence in the detection result from a credible model. Meanwhile, if many models with relatively low $p_k(\theta)$ generate abnormal residuals at the same time, the score $s(t)$ will also be high. In this case, the system is likely to be faulty too because model uncertainties should not affect many residuals simultaneously, especially if some of the residuals are logically independent and characterize systems from different perspectives.

Based on the same reason, we can also replace the fault isolation (16) with the following equation:

$$C_f = \arg max_j V_j, \quad where \quad V_j = \frac{\sum_{i=1}^{n}(r_i(t) \cap C_{ij}) \cdot p_k(\theta_i)}{\sum_{i=1}^{n}(r_i(t) \cup C_{ij}) \cdot p_k(\theta_i)}. \tag{20}$$

Basically, the above equation implies that it is more important to correlate the residuals $r_i(t)$ from high quality models than those from low quality models because the residuals from high quality models are more credible.

7 Model Search Experiments

In this section, we design several experiments to verify whether invariant models widely exist in real systems. Our experiments are performed in a testbed system with a typical three-tier architecture, as shown in Fig. 1. The system includes an Apache Web server, a JBoss application server, and a MySQL database server. The application software running on this system is Pet Store [23]. Pet Store is a sample application written by Sun Microsystems to demonstrate how to use the J2EE platform in developing flexible, scalable, cross-platform e-commerce applications. Pet Store includes 27 Enterprise Java Beans (EJBs), some Java Server Pages (JSPs), and Java Servlets.

Just like other Internet services, here, users can visit the Pet Store Web site to buy various pets. We develop a client emulator to generate a large class of different user scenarios and workload patterns. The number of users emulated at a time point and the duration for emulating that number of users are both randomly generated with a big variance. Therefore, each run of experiments always results in a different workload. 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. A certain randomness of user behavior is also considered in the emulated workload. For example, a user action is randomly selected from all possible user scenarios that could follow the previous user action. The time interval between two user actions is also randomly selected from a reasonable range.

Monitoring data are collected from the three servers used in our testbed system. Fig. 12 shows the categories of monitoring data used in our experiments. In total, we have eight categories and each category includes a different number of measurements. The number of measurements in each category is given in the right three columns, which represent three servers, respectively. Note that the measurements in the "JVM" and "JBoss" category are collected through Java Management Extensions (JMX) [10].

These monitoring data are used to calculate flow intensities and the sampling time is 10 seconds. In total, 111 flow intensity measurements are derived from the above monitoring data. Though these measurements are collected from various points across systems and have different physical meanings, many of them have surprisingly similar curves as time series. This is because many measurements react to the same workload accordingly. Note that the workload itself changes significantly here and includes many dynamics. Though some measurements may include noisy spikes in their curves, they have very similar long-run trends and mass characteristics.

Category	Measurements	Web	AP	DB
CPU	utilization, user usage time, system usage time, idle time, IO wait time, IRQ time, soft IRQ time	7	7	7
Disk	# of write operations, # of write sectors, # of write merges, write time, IO time, # of pending IO operations, weighted IO time	7	7	7
OS	# of used file descriptors, # of max file descriptors, free physical memory size	2	3	2
Network	# of RX packets, # of RX bytes, # of RX errors, # of RX multicasts, # of TX packets, # of TX bytes	11	6	6
JVM	used heap memory size, processing time, # of live threads, peak # of threads, total # of threads	-	5	-
JBoss	# of processing EJBs, # of cached EJBs, # of pooled EJBs, # of created EJBs, # of DB connections, # of DB connections in use	-	6	-
Apache	# of http requests, # of http requests based on specific URL types	13	-	-
MySQL	# of SQL queries, # of SQL queries based on specific table types	-	-	15

Fig. 12. Categories of measurements.

We manually check the curves of all 111 flow intensity measurements and observe that 74 measurements have curves similar to the workload curve. This means that they should have strong linear relationships, which can be well described with the ARX model. The remaining 37 measurements either do not respond to the intensity of workloads or respond in an unknown way. However, these measurements may still have linear correlation with other local measurements.

As described in Algorithm 6.1 (Fig. 10), we first construct a model for each pair of measurements and then sequentially test the validity of these models with new data. In our experiments, we collect 1.5 hours of data to construct models and then continue to test these models for every half an hour, i.e., the window size is half an hour. Note that since the sampling time is 10 seconds, half an hour monitoring data includes 180 sampling points for each measurement. Studies have shown that users are often dissatisfied if the latency of their Web requests is longer than 10 seconds. Therefore, the order of the ARX model (i.e., $[n, m, k]$ shown in (1)) should have a very narrow range. In our experiments, since the sampling time is 10 seconds, we let $0 \leq n, m, k \leq 2$. Given two flow intensity measurements, logically we do not know which one should be chosen as the input or output (i.e., x or y in (1)) in complex information systems. Therefore, in our experiments, we construct two models (with reverse input and output) first but only choose the model with higher fitness score to characterize the correlation between two measurements.

By combining every two measurements among the total of 111 measurements, Algorithm 4.1 (Fig. 7) in total builds 6,105 models as invariant candidates. For each model, a fitness score is calculated according to (8). In our experiments, we select the threshold of fitness score $\widetilde{F} = 50$. According to (17), a model is considered as valid only if its fitness score is higher than 50. In fact, if a model's fitness score is over 50 (a relatively high score), we observe that there always exists strong linear correlation between its two measurements. We observe that 1,158 models among the total of 6,105 models have fitness scores higher than this threshold.

As illustrated in Algorithm 6.1 (Fig. 10), these 1,158 models are chosen as valid invariant candidates for the following sequential testings. In our experiments, we set the threshold of confidence scores $P = 1$, i.e., a model will

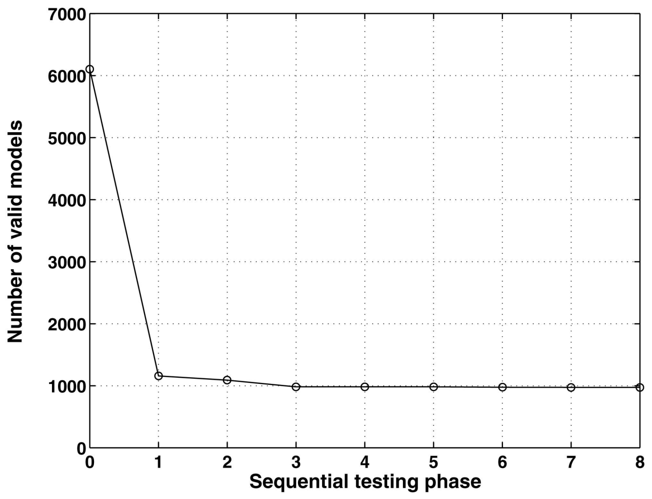


Fig. 13. Number of valid models in testings.

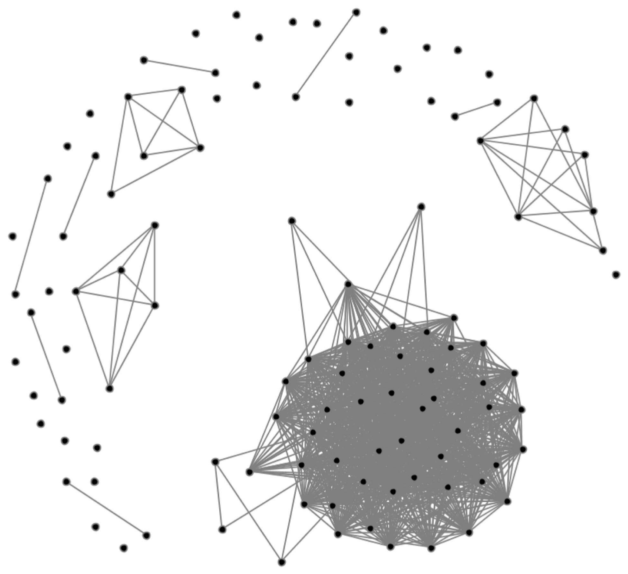


Fig. 14. Network of measurements and invariants.

be immediately discarded if it fails the validity test in one time window. As mentioned earlier, we use thoroughly different workloads to generate the monitoring data for sequential testings and each testing phase includes half an hour data. Fig. 13 shows how many models are discarded in each phase. From the figure, we can see that the number of valid models becomes stable quickly after several testing periods. Many unstable models are removed in the first phase because we choose a relative high fitness score threshold. After seven phases of sequential testings with various workloads, eventually, we end with 975 valid models. Due to unexpected system dynamics and uncertainties, some of these models might become invalid if we never stop the testing process. However, as shown in Fig. 13, this number seems to be quite stable. In the following analysis, without loss of generality, we consider these 975 models as likely invariants because theoretically a model can be regarded as an invariant only if the model holds all the time.

We observe that the models with low fitness scores in the first phase are usually not robust and they are likely to be discarded in the following sequential testing process. These models are liable to be affected by various system dynamics and uncertainties. Conversely, those models with high fitness scores in the first phase are very robust through all testing phases. This means that the fitness score given in (8) is a good metric for evaluating relationships between measurements.

With 111 flow intensity measurements, we eventually discover 975 likely invariants from them. Fig. 14 illustrates how these invariants are distributed across these measurements. This figure is plotted with an open source software library named JUNG [24]. In the figure, a node represents a measurement while an edge represents an invariant relationship between the connected two nodes. Therefore, we have 111 nodes and 975 edges in this figure. Among 111 measurements, 29 measurements are the isolated nodes and they do not have strong relationships with any other measurements. In the figure, there is a meshed "core" which consists of 51 measurements. As mentioned earlier, we manually discover that 74 measurements among the total of 111 measurements have very similar evolving curves corresponding to the workloads. The 51 nodes in the meshed "core" are a subset of those 74 measurements while the other 23 measurements are too noisy to be included in this core, though they all respond to the workloads directly. Therefore, these 51 measurements have many relationships between each other. There are many small clusters and lines in the figure, which include 31 measurements. As mentioned earlier, they are local invariants that characterize local relationships between measurements. For example, the measurement "disk write time" has a linear correlation with another local measurement "disk weighted IO time" though they both respond to the workloads in an unknown way. Such local relationship is reflected with an isolated line in the figure. A large number of global and local invariants could be combined together to effectively characterize large, dynamic, and complex systems. As shown in Fig. 14, essentially, we add a "virtual network of invariants" on physical systems, which keeps monitoring the activities of various system components.

8 Fault Detection Experiments

In Section 4, we have given an example of a memory leaking fault to illustrate the effectiveness of our approach. As discussed earlier, various faults could affect a complex system in very different ways. In this section, we analyze a list of faults to demonstrate that our approach is able to detect a wide class of faults. Note that our approach is essentially anomaly detection, which does not use specific signatures of faults for detection. The list of injected faults are only used as examples to demonstrate the feasibility of our approach, which could detect other faults as well. In practice, fault injection itself is a difficult task [25]. In our experiments, we try to make each fault as realistic as possible. However, some runtime faults are hard to inject so we add specific logic as a failure in the source code to simulate their impacts on system behavior rather than their breakout mechanisms.

In our experiments, we inject various failures into the system to simulate faults and then use the collected monitoring data to compute residuals. As discussed in Section 4, we determine whether the system is faulty based on the abnormality of these residuals. We demonstrate a fault's impact in the following two ways. At first, in all our experiments, we choose the models described in (9), (10),

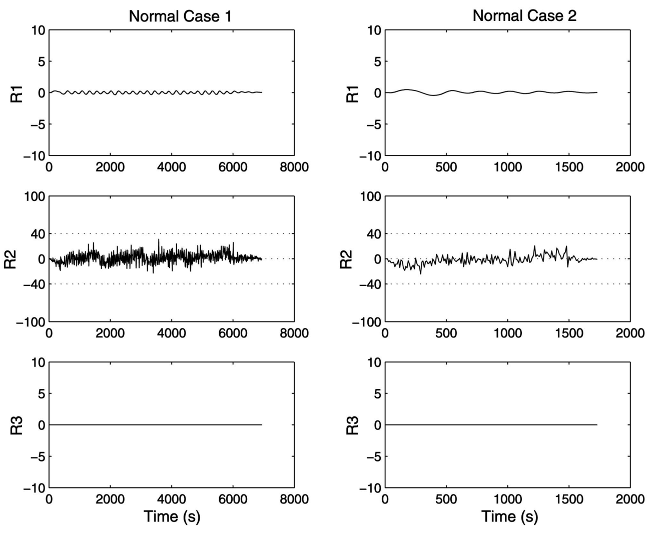


Fig. 15. Two normal cases.

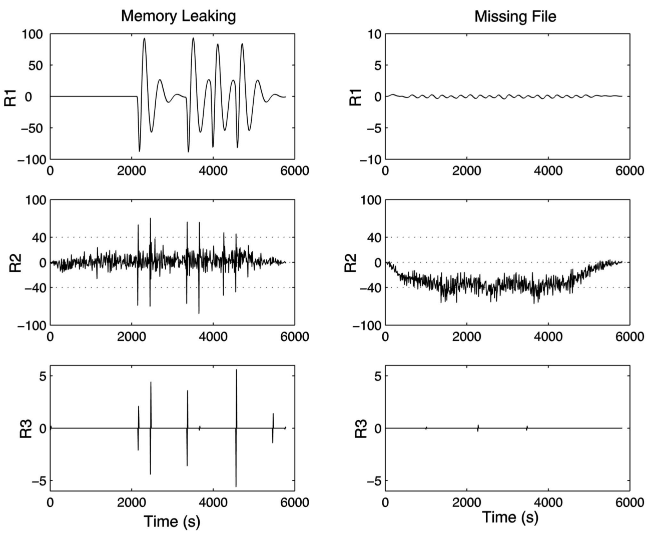


Fig. 16. Memory leaking and missing file.

and (11) to generate residuals and then we interpret the changes of these three residuals in a microscopic view. Second, in all our experiments, we track the changes of all 975 invariants extracted from the above experiments to take a macroscopic view on a fault's impact. Note that all invariants, including the three equations used in the microscopic view, are tracked in the same way and we only use these three residuals as an illustration. The same faults are also injected into various components with different intensity to demonstrate the robustness of our approach. 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 mentioned earlier, though flow intensity measurements (such as x and y) change all the time, the validity of invariants (the equation $y = f(x)$) are not affected by varying workloads.

8.1 Normal Cases

In Section 4, Fig. 6 has shown three residuals of a normal case (i.e., a case without fault injection). Many other normal cases (10 cases) are used to verify whether these models capture flow dynamics well under different workloads. According to (12), the same three residuals are generated in each of these normal cases. Fig. 15 illustrates the typical residuals resulting from normal situations. Comparing the residuals in these normal cases, we notice that the variances of $R1$ and $R3$ are almost close to zero. The curve of $R2$ is much noisier and the largest variance of $R2$ is close to 35. For simplicity, we set the threshold of $R2$ equal to 40 in our detection experiments. There are no significant large residuals in all these figures. All residuals have mean values close to zero so that they seem to be stationary. This means that the models described in (9), (10), and (11) capture flow dynamics well in various normal situations. Note that in all our experiments, we use the same three models to illustrate the impact of faults though workloads are different. The horizontal axis of all figures represents time in seconds. Since we use different workloads in our experiments, the duration of workloads varies in different figures. $R1$, $R2$, and $R3$ refer to the residuals resulting from CPU usage (percentage), number of SQL queries and number of HTTP requests per second, respectively.

We also track the changes of 975 invariants in our experiments. Among 10 normal cases, we observe that 8 cases have one broken invariant (i.e., the residual of one invariant is over its threshold) while the other 2 cases have 6 and 4 broken invariants, respectively. As discussed in Section 6, we can compute $s(t)$ based on (19) to measure the abnormality of system behavior. Since we set the threshold $P = 1$ to extract invariants in Section 7, those models with $p_k(\theta) < 1$ are discarded in the sequential testing process and the left 975 invariants have $p_k(\theta) = 1$. Given $n = 975$ and $p_k(\theta) = 1$ in (19), $s(t)$ is solely determined by the number of broken invariants. Therefore, in the following sections, we use the number of broken invariants to illustrate the impact of faults. If we choose a threshold $S < 6/975$, the broken invariants in these cases could lead to false positives in detection. We extract 975 invariants after several sequential testing phases but, in practical operation, we should never stop the sequential testing process. It seems that some of these likely invariants are not stable and do not always characterize system dynamics well under various uncertainties.

8.2 Memory Leaking

Memory leaking is a common software bug where 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 a memory leaking bug could run correctly for a long period of time before it eventually causes something serious 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 [26].

We inject this failure into "ShoppingCart" EJB of Pet Store software by repeatedly allocating 10K heap memory and making it reachable from a long-lived object. For comparison convenience, Fig. 16 includes the same figures shown in the right column of Fig. 6. As analyzed in Section 4, memory leaking periodically leads to the intensive operation of JVM garbage collection because of unreleased objects in heap memory. Essentially, this operation affects the flow dynamics captured by those three models. For example, many requests are delayed during the time of this operation and users will notice slow response of this Web site. All three residuals show strong signals revealing the existence of a fault.

We inject the same failure into several other EJBs of Pet Store software and track the changes of invariants in each case. These EJBs are randomly selected for fault injection. Each experiment only runs half an hour so that memory leaking is not severe. The number of broken invariants resulting from each case is listed after the fault-injected EJB names: AddressEJB (5), CatalogEJB (17), and ShoppingCart EJB (21). As discussed earlier, if we choose the threshold $S = 6/975$, we will have one false negative because AddressEJB is not frequently used in user requests and not much memory leaked in this case. The effectiveness of our approach is also dependent on the resolution of available monitoring data and the number of extracted invariants related to a specific fault. Note that if we use a different set of monitoring data, we will extract a different set of invariants and these faults could lead to a much different number of broken invariants.

8.3 Missing File

This type of fault could result from an operator's occasional mistakes during system maintenance and management. For example, while an operator is modifying a configuration file of Web applications, he mistakenly deletes a JSP file in the same folder. Assume that a specific HTTP request includes visits to this missing JSP file as well as several other JSPs. Due to the successful return of other JSPs, the Web server may regard this HTTP request as successfully completed and put a "HTTP 200" record in its access log, i.e., in this case, the exception from the middleware may not be propagated back to the Web server. Therefore, operators may not notice such a fault in practice.

We inject such a failure into the Pet Store application by deleting "mylist.jsp" file. After a user logs into his account, this JSP presents a list of his favorites profiled before. Users will not see this portion of content on the Web page after the deletion of this file. The right column of Fig. 16 shows the residuals generated from this case. Both $R1$ and $R3$ seem to be very normal. The curve of $R2$ shows a strong abnormal pattern. Unlike the "peaks" in the memory leaking case, here, the value of $R2$ is negative and large all the time. This reveals that some requests stopped in the middleware and did not invoke SQL queries as usual. This is true because "mylist" JSP includes SQL queries to retrieve a user's favorites from the database. Therefore, $R2$ can help us to detect such faults. As discussed earlier, besides the residual threshold, we should also use the general patterns of residuals to detect and isolate faults. As exemplified by this case, both the shape of residual curves and the mean of residuals include useful information about the type of faults. Memory leaking fault delays user requests but eventually all user requests are completed. Therefore, the mean of $R2$ is close to zero although $R2$ includes some "peaks."

In the same way, we manually delete several other JSP files to track the changes of all invariants. The number of broken invariants is listed after its related JSP file name: product.jsp (1), category.jsp (2), item.jsp (4), mylist.jsp (88), and cart.jsp (238). From these numbers, we observe that the impact of this fault varies in a big range. Some JSP files are much more frequently used than others in user requests. This fault (deleting a JSP file) also does not affect those performance-related measurements at all. Based on the monitoring data given in Fig. 12, our approach is only able to detect those faults with severe impacts. Since flow intensity and flow dynamics capture the mass characteristics of user requests, in general it is difficult to detect those faults with small and weak impacts.

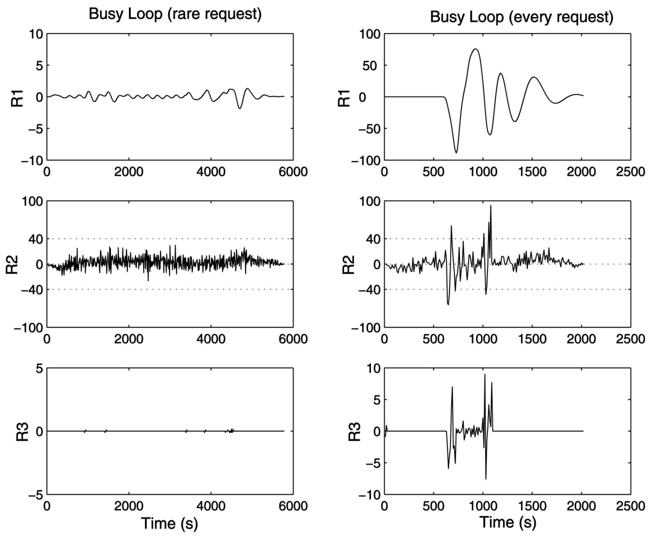


Fig. 17. Busy loop.

8.4 Busy Loop

Infinite loop is another notorious software bug that could be made by novices as well as experienced programmers. Typically, unexpected behavior 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 certain period of time. With regard to the impact of faults on the system, two types of "busy loop" failures are used in our experiments. The first type only affects a very small number of user requests. In our experiment, we choose one specific user request "update.do" to inject this failure. Every time when this specific request is submitted to the system, it will enter a busy loop *with 0.001 probability*. Conversely, the second type of "busy loop" failure has a much more serious impact. When this specific request is submitted, it will *always* enter a busy loop. In both cases, the busy loop only lasts for very short time.

Fig. 17 shows the residuals from both types of failures. For the first type of "busy loop" failure (the left column), $R2$ and $R3$ seem to be very normal because the busy loop takes a very short time and will not affect much the latency of user requests. However, the curve of $R1$ is abnormal. $R1$ is almost close to zero in normal cases and, here, we see some small "peaks" in the curve. This is because the injected busy loops increase the CPU usage. For the second type of fault (the right column), all three residuals show strong signals of the fault's existence. Comparing these curves with those in the memory leaking case, we notice that they have very different shapes. In the memory leaking case, we only see some "peaks" rather than the distorted segments here. This is because the intensive garbage collection operations only occur once in a while (not continuously). Each time after an operation is completed, the system quickly goes back to the normal situation and, therefore, we only see "peaks" periodically. In the second busy loop case, every request is affected during a period of time and therefore segments of residuals are distorted. This is another example where we can use the shape of residual curves to isolate faults. System

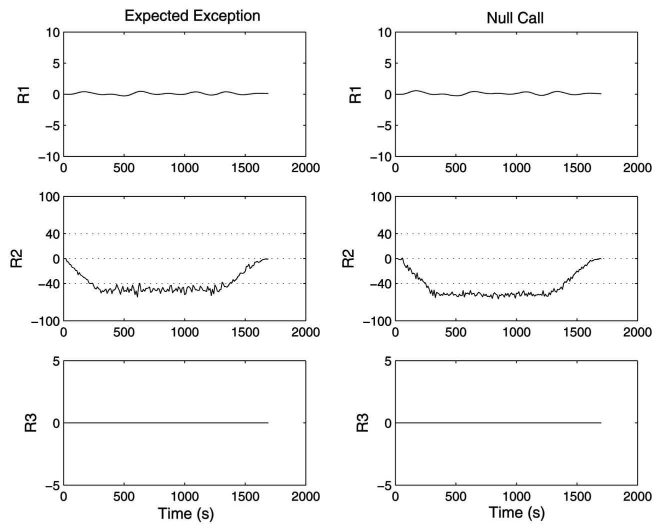


Fig. 18. Expected exception and null call.

administrators may develop such knowledge incrementally during their system management practice.

Since the second type of busy loop fault always causes severe impacts on system behavior, we repeat the first type of fault injection several times with different intensity to track its impact on invariants. A different number of busy loops are injected into that specific user request and the number of broken invariants is listed as the following: 30 loops (1), 65 loops (5), 100 loops (110), 150 loops (117), and 300 loops (133). As shown in Fig. 12, since we use many performance-related measurements, the busy loop fault is likely to affect these measurements and cause many invariants to be broken.

8.5 Expected Exception and Null Call

"Expected exception" and "null call" failures proposed in [27] are also used in our fault injection experiments. "Expected exception" failure is injected into components that contain methods which declare exceptions. After the expected exception failure is injected into a component, any invocation of its methods declaring exceptions will raise the declared exception immediately (if the method declares many exceptions, an arbitrary one is chosen and thrown). However, methods in that component which do not declare exceptions are unaffected by this injected failure. Meanwhile, "null call" failures can be injected into any component in the Pet Store application. After a null call failure is injected into some component C, any invocation of a method in component C results in an immediate return of a null value (i.e., calls to other components are not made). These bugs can easily happen in practice due to incomplete, or incorrect, handling of rare conditions. In our experiments, we randomly select the "Address" EJB for expected exception injection and the "Catalog" EJB for null call injection.

Fig. 18 shows the residuals generated from these two cases. In both cases, $R1$ and $R3$ are normal but $R2$ is very abnormal. In the figure, the value of $R2$ is large and negative all the time. As discussed earlier, the shape of $R2$ implies that some user requests stopped in the middleware and did not invoke SQL queries as usual. In fact, this can be well explained by the mechanism of these two failures.

We inject both faults into different components and track the changes of invariants in each case. The "expected exception" failure is injected into "AddressEJB" and "AccountEJB," which results in 363 and 316 broken invariants, respectively. Meanwhile, the "null call" failure is injected into "CatalogEJB" and "AddressEJB," which leads to 283 and 22 broken invariants, respectively. Since many flow intensity measurements are derived from various HTTP requests and SQL queries, many invariants are extracted from these measurements for their relationships. As discussed above, the "expected exception" and "null call" failures break many such invariants.

8.6 Discussion

Based on the above fault detection experiments, we notice that various residuals are effective in detecting various faults. This is because flow intensities at various points have different physical meanings and invariants also characterize systems from many different perspectives. The different symptom combination of residuals could also help us to distinguish and isolate faults. We may also gain some insights into the nature of faults based on specific patterns shown in residuals. As discussed in Section 4, with more such invariants, we should be able to detect a wider class of faults in our framework. Note that the number of broken invariants listed in each case is meaningful for feasibility analysis rather than performance evaluation. As mentioned earlier, if we use a different set of monitoring data, we should extract a different set of invariants and each injected fault could lead to a much different number of broken invariants. In addition, our approach is essentially an anomaly detection and the impacts of various faults are much dependent on how the extracted invariants are related to a fault.

To the best of our knowledge, we believe that we proposed the first general solution/framework that is able to detect and isolate faults across all sections of distributed information systems. In addition, our flow and invariant-based analysis enables us to detect and isolate a variety of faults including operator mistakes, software bugs, networking problems, and hardware faults. Much prior work only introduced fault detection and isolation mechanisms for specific components but not for the whole complex system and for such a variety of faults. For example, Kiciman and Fox [27] proposed tracing user requests in application servers and detect faults by checking the shape of EJB calling sequences. While their instrumentation of middleware enables them to detect and isolate some faults at a fine granularity (the EJB component level), their approach is only designed to detect faults in the application software running on middleware. It's not clear how to collect user request traces from other components (such as databases) and other distributed systems without such middleware. Meanwhile, many faults such as memory leaking and operator errors may not affect the shape of use request traces at all.

Conversely, our approach is designed to manage all sections of distributed systems at a relative coarse granularity. For example, based on the relationship between the number of SQL queries and the volume of incoming network packets to a database server, we can monitor and track the activity of the database server. We can monitor a disk's activity by tracking the invariant relationship between local measurements such as "disk write time" and "disk weighted IO time." We can also monitor the internal activity of databases by tracking the relationships between the intensities of different types of SQL queries. In fact, the three-tier Web system is only used as an example to demonstrate our framework. Our approach is not dependent on any specific properties of such systems and should work for many other distributed transaction systems as well. As mentioned earlier, we only use lightweight and commonly available monitoring data and do not instrument systems in any specific ways. Therefore, for a specific component such as the application software running on middleware, our approach may not detect and isolate faults at the same granularity as those methods proposed in [27] do. Basically, it's not feasible to monitor and track complex systems with large scope as well as fine granularity at the same time, especially for complex systems with hundreds of servers.

The effectiveness of our approach is strongly dependent on the resolution of monitoring data from real systems. If we cannot collect monitoring data from some components, we cannot derive flow intensities and then apply our approach to track these components. Meanwhile, since flow intensity and flow dynamics capture the mass characteristics of user requests, it is difficult to detect those faults with small and weak impacts. Some faults may only be detected after they cascade and evolve to more significant faults. In addition, our flow and invariant-based approach may not work well in systems with a very limited number of invariants. Essentially, whether a fault can be detected is dependent on the existence of its related invariants.

Due to the lack of benchmark system and data, it is difficult to quantitatively compare our approaches with prior work. We collect thoroughly different types of monitoring data for fault analysis, and detection accuracy is also much dependent on the resolution of available monitoring data. Our approach is also proposed to manage all sections of distributed systems rather than a specific component. There are also no standard implementation of systems, faults, workloads, etc., for us to evaluate the effectiveness of different approaches.

9 Related Work

Detection and diagnosis of faults in complex information systems is a formidable task. Most of the current approaches for fault diagnosis use event correlation [28]. This method collects and correlates events to locate faults based on known dependency knowledge between faults and symptoms. In practice, many runtime faults in an interconnected system are not anticipated and well understood. Meanwhile, runtime environments are so diverse that a fault may manifest itself in various ways. As a result, it is usually difficult to obtain such fault-symptom dependency knowledge precisely in complex systems. Essentially, our approach is based on anomaly detection. We develop novel concepts and approaches to model transaction flow dynamics of the monitored system and further track the change of flow dynamics for fault detection. We use residuals to interpret the monitoring data effectively.

The Berkeley/Stanford 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 [3]. Based on trace analysis, another failure detection mechanism was recently discussed in [29]. As discussed in Section 2, with regard to the huge volume of user visits, it is difficult to monitor, collect, and analyze each individual request. Additionally, current methods of collecting traces result in a large monitoring overhead and also need to modify the middleware. In this paper, we analyze the mass characteristics of user requests and further model the dynamic property of complex systems for fault detection. Our approach relies on flow intensity measurements extracted from widely available and lightweight monitoring data such as log files. Therefore, our approach introduces a practical fault detection mechanism for Internet services as well as for other distributed transaction systems. In addition, our approach is proposed to detect and isolate a variety of faults in all sections of distributed systems rather than a specific component. Aguilera et al. [30] proposed two algorithms to isolate performance bottlenecks in distributed systems composed of black-box nodes. Their data instrumentation and monitoring methods can be used in our work to derive flow intensity measurements at a fine granularity.

The model-based fault detection and isolation method has been widely analyzed in control theory. However, this method has been mostly applied in fault detection of mechanical and electronic systems [31], where analytical models are not difficult to derive from first principles such as physical laws. To the best of our knowledge, we have not seen its application in complex information systems. One difficulty is to derive meaningful models in large and complex information systems. In this paper, the novel concepts of flow intensity and flow dynamics enable us to learn analytical models from monitoring data and further use these models in model-based fault detection.

10 Conclusions

In this paper, we introduce novel concepts, flow intensity and flow dynamics, to characterize the dynamics of mass user requests and further model the invariant property of complex systems. We calculate flow intensity measurements from commonly available and lightweight monitoring data. A system identification method is proposed to model transaction flow dynamics between flow intensity measurements collected at various points. With the learned model, a model-based FDI approach is proposed to track the changes of invariants. The occurrence of faults could affect the usual flow dynamics in distributed systems. Therefore, by tracking a set of invariants underlying the system, we are able to detect a wide class of faults. We also proposed an algorithm to automatically search and validate relationships between flow intensity measurements. Our algorithm enables systems to have a certain level of self-cognition capability, which is especially desirable for complex system management.

For simplicity, we only employed an ARX model as an example to illustrate our framework. In fact, many other models can also be used to characterize a multivariate data relationship. For example, we have used a Gassuian mixture model to statistically learn the joint density distribution of multiple flow intensity measurements and further regard this distribution as an invariant of the monitored system. Using the same framework, we notice that this model is also effective in detecting some faults. In our future work, we will develop a library to include various multivariate data models so as to characterize various dynamic relationships among flow intensity measurements. We also want to explore whether the mature tracking technology such as the Kalman filter [32] can be applied to reduce noise in flow dynamics tracking.

References

[1] D. Patterson, "A Simple Way to Estimate the Cost of Downtime," *Proc. 16th System Administration Conf. (LISA '02),* pp. 185-188, 2002.

[2] D. Oppenheimer, A. Ganapathi, and D. Patterson, "Why Do Internet Services Fail, and What Can Be Done About It," *Proc. Fourth Usenix Symp. Internet Technologies and Systems (USITS '03),* 2003.

[3] M. Chen, A. Accardi, E. Kiciman, J. Lloyd, D. Patterson, A. Fox, and E. Brewer, "Path-Based Failure and Evolution Management," *Proc. First USENIX Symp. Networked Systems Design and Implementation (NSDI '04),* Mar. 2004.

[4] http://www-306.ibm.com/software/tivoli/, 2005.

[5] http://www.openview.hp.com, 2005.

[6] http://www.smarts.com, 2005.

[7] R. Isermann, "Model-Based Fault Detection and Diagnosis—Status and Applications," *Proc. 16th IFAC Symp. Automatic Control in Aerospace (ACA '04),* June 2004.

[8] J. Gertler, *Fault Detection and Diagnosis in Engineering Systems.* Marcel Dekker, 1998.

[9] http://phx.corporate-ir.net/phoenix.zhtml?c=97664&p=irol-newsArticle&ID=7989% 60&highlight=, 2005.

[10] http://java.sun.com/products/JavaManagement/, 2005.

[11] http://manageengine.adventnet.com/, 2005.

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

[13] R. Redner and H. Walker, "Mixture Densities, Maximum Likelihood and the Em Algorithm," *SIAM Rev.,* vol. 26, no. 2, pp. 195-239, 1984.

[14] H. Akaike, "Information Theory and an Extension of the Maximum Likelihood Principle," *Proc. Second Int'l Symp. Information Theory,* 1973.

[15] J. Rissanen, "Prediction Minimum Description Length Principles," *Ann. Statistics,* vol. 14, 1986.

[16] http://www.jboss.org, 2005.

[17] M. Spiegel, *Theory and Problems of Probability and Statistics.* McGraw-Hill, 1992.

[18] J. Han and M. Kamber, *Data Mining: Concepts and Techniques.* Morgan Kaufman, 2000.

[19] S. Zhang, I. Cohen, M. Goldszmidt, J. Symons, and A. Fox, "Ensembles of Models for Automated Diagnosis of System Performance Problems," *Proc. Int'l Conf. Dependable Systems and Networks (DSN),* pp. 644-653, 2005.

[20] I. Cohen, S. Zhang, M. Goldszmidt, J. Symons, T. Kelly, and A. Fox, "Capturing, Indexing, Clustering, and Retrieving System History," *ACM SIGOPS Operating Systems Rev.,* vol. 39, no. 5, pp. 105-118, 2005.

[21] N. Cristianini and J. Shawe-Taylor, *An Introduction to Support Vector Machines and Other Kernel-Based Learning Methods.* Cambridge Univ. Press, 2000.

[22] R. Yager, M. Fedrizzi, and J. Kacprzyk, *Advances in the Dempster-Shafer Theory of Evidence.* Wiley, 1994.

[23] http://java.sun.com/developer/releases/petstore/, 2005.

[24] J. O'Madadhain, D. Fisher, S. White, and Y. Boey, "The Jung (Java Universal Network/Graph) Framework," Technical Report UCI-ICS 03-17, Univ. of California at Irvine, School of Information and Computer Sciences, jung.sourceforge.net, 2003.

[25] J. Voas and G. Mcgraw, *Software Fault Injection: Inoculating Programs against Errors.* John Wiley & Sons, 1997.

[26] B. Tate, *Bitter Java.* Manning Publications, 2002.

[27] E. Kiciman and A. Fox, "Detecting Application-Level Failures in Component-Based Internet Services," *IEEE Trans. Neural Networks,* vol. 16, no. 5, pp. 1027-1041, 2006.

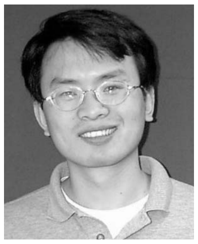
[28] A. Yemini and S. Kliger, "High Speed and Robust Event Correlation," *IEEE Comm. Magazine,* vol. 34, no. 5, pp. 82-90, May 1996.

[29] G. Jiang, H. Chen, C. Ungureanu, and K. Yoshihira, "Multi-Resolution Abnormal Trace Detection Using Varied-Length ngrams and Automata," *Proc. Second IEEE Int'l Conf. Autonomic Computing (ICAC '05),* June 2005.

[30] M. Aguilera, J. Mogul, J. Wiener, P. Reynolds, and A. Muthitacharoen, "Performance Debugging for Distributed Systems of Black Boxes," *Proc. 19th ACM Symp. Operating Systems Principles (SOSP '03),* pp. 74-89, 2003.

[31] R. Isermann and P. Balle, "Trends in the Application of Model-Based Fault Detection and Diagnosis of Industrial Process," *Control Eng. Practice,* vol. 5, no. 5, 1997.

[32] R. Kalman, "A New Approach to Linear Filtering and Prediction Problems," *Trans. ASME-J. Basic Eng.,* vol. 82, no. series D, pp. 35-45, 1960.



Guofei Jiang received the BS and PhD degrees in electrical and computer engineering from Beijing Institute of Technology, China, in 1993 and 1998, respectively. During 1998-2001, he was a postdoctoral fellow in computer engineering at Dartmouth College, New Hampshire. He is currently a research staff member with the Robust and Secure Systems Group in NEC Laboratories America, Princeton, New Jersey. During 2001-2004, he was a research scientist with the Institute for Security Technology Studies at Dartmouth College. His current research focus is on distributed systems, dependable and secure computing, and system and information theory. He has published more than 50 technical papers in these areas. He is an associate editor of *IEEE Security and Privacy* and has served on the program committees of many conferences. He is a member of the IEEE.

Haifeng Chen received the BEng and MEng degrees, both in automation, from Southeast University, China, in 1994 and 1997, respectively, and the PhD degree in computer engineering from Rutgers University, New Jersey, in 2004. He has worked as a researcher with the Chinese National Research Institute of Power Automation. He is currently a research staff member with NEC Laboratories America, Princeton, New Jersey. His research interests include data mining, autonomic computing, pattern recognition, and robust statistics.

Kenji Yoshihira received the BE degree in electrical engineering from the University of Tokyo in 1996 and the MS degree in computer science from New York University in 2004. He designed processor chips for enterprise computer at Hitachi Ltd. for five years. He employed himself in CTO at Investoria Inc. in Japan to develop an Internet service system for financial information distribution through 2002. He is currently a research staff member with the Robust and Secure Systems Group in NEC Laboratories America, Princeton, New Jersey. His current research focus is on distributed systems and autonomic computing.