Network-based Problem Detection for Distributed Systems

Hisashi Kashima Tadashi Tsumura Tsuyoshi Idé Takahide Nogayama Ryo Hirade
Hiroaki Etoh Takeshi Fukuda
IBM Tokyo Research Laboratory
1623-14 Shimo-tsuruma, Yamato-shi, 242-8502 Kanagawa, Japan.
{hkashima, dashi, goodidea, nogayama, rhirade, etoh, fukudat}@jp.ibm.com

Abstract

We introduce a network-based problem detection framework for distributed systems, which includes a data-mining method for discovering dynamic dependencies among distributed services from transaction data collected from network, and a novel problem detection method based on the discovered dependencies. From observed containments of transaction execution time periods, we estimate the probabilities of accidental and non-accidental containments, and build a competitive model for discovering direct dependencies by using a model estimation method based on the online EM algorithm. Utilizing the discovered dependency information, we also propose a hierarchical problem detection framework, where microscopic dependency information is incorporated with a macroscopic anomaly metric that monitors the behavior of the system as a whole. This feature is made possible by employing a network-based design which provides overall information of the system without any impact on the performance.

1 Introduction

Today's typical e-business infrastructure consists of distributed heterogeneous components such as web servers, application servers, database servers, storage servers, and network devices, each of which provides various different functions as *services*. The services have complex interdependency that one service uses others to fulfill its functions. For example, a Web server provides clients with Web pages as services each of which is identified with a universal resource locator (URL). When a user requests a Web page on the server using a Web browser, then the server receives an HTTP request for the page. If the requested page contains a servlet, the Web server calls an application server to execute the servlet. Therefore we say that request for the Web page depends on the servlet. Likewise, if the servlet calls an EJB, the servlet depends on the EJB, and if the EJB issues queries to a database, the EJB depends on the database. Figure 1 shows an example.

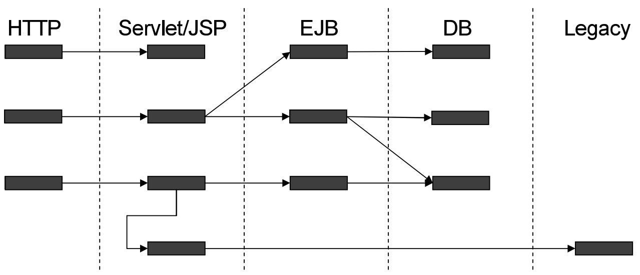

Figure 1. Example of dependencies in a distributed computer system

The dependency information can be considered as a graph in which services are represented as nodes and dependencies among services are represented as arcs. The dependency information is useful for understanding system behaviors that are sometimes not noticed by system administrators. Also in the case of system failures, it is useful for failure detection, impact analysis and root cause analysis. When one service goes wrong because of hardware malfunctions or software bugs, the dependency information tells us impacted services that will not work correctly due to the problem. In the previous example, if some error occurred during the execution of a database query, from the dependency information we can see that the Web page that indirectly depends on the database query would not be shown to the client correctly. Inversely when a user observes a problem at a Web page and the Web server does not have any problem, then at least one of the services that the Web page directly or indirectly depends on must have the root cause. There are several papers on problem determination and root cause analysis using dependency graphs [27, 8, 12].

It is very difficult even for the system designers and system administrators to totally understand the dependency information in a distributed computer system, since many components dynamically bind with each other at runtime, and the dependencies often change over time as the environment changes. Therefore it is natural to consider that data mining techniques can be useful to discover dynamic dependencies from the running computer system.

In [13], Gupta *et al.* addressed the problem of automatically discovering dependencies among services from historical execution time period data collected by common performance monitoring instrumentation that many existing components are already equipped with. They presented an on-line algorithm that efficiently discovers all dependencies from a given execution time period data. However, the algorithm may generate many false positives as workload becomes higher. In addition, their method does not distinguish direct dependencies and indirect dependencies. For instance in Figure 1, although HTTP requests do not call EJBs directly, their method extracts this indirect relation implied by the chain of calls as a dependency. This property complicates dependency graphs, which is an obstacle to understand the target system, and makes root cause analysis more difficult. In contrast, direct dependencies form a very simple graph, and therefore we claim that direct dependencies are more suitable for these purposes.

In this paper, we propose a method that solves the above problems by using two ideas. First, we mitigate false dependencies by introducing a statistical model that takes into account of false positive rates estimated from the arrival times and the response times of services.

Next, we propose a competitive model for inferring the parent service that directly calls a service, and a model estimation method based on EM algorithms [9]. Moreover our method can estimate the number of times a particular service *directly* calls another service. This is a generalization of the task defined by Gupta *et al.* that aims to infer if there exists a dependency between two services.

Utilizing the dependency information, we also propose a hierarchical fault detection framework, where microscopic dependency information is incorporated with a macroscopic anomaly metric that monitors the behavior of the whole system. This feature is made possible by employing a network-based design which can provide overall information of the system without any impact on the performance.

The rest of the paper is organized as follows. Section 2 gives preliminaries necessary for describing the problem and our approaches to it. Section 3 describes the network-based architecture of our monitoring system. Section 4 outlines Gupta's algorithm and points out its limitations. Section 5 gives an improved method for estimating strength of dependencies. Section 6 describes our anomaly detection framework. Section 7 experimentally evaluate our dependency discovery algorithm and demonstrate the utility in detecting application faults. Section 8 refers to related work. Finally, Section 9 concludes the paper.

2 Dependency discovery problem

In Section 1, we motivated ourselves to use dependency information to detect and analyze system failures. In this section, we define the problem of finding dependencies from running systems.

Let Σ be a set of *services* of interest in a computer system. $S \in \Sigma$ represents a service such as a Web page, a servlet, an EJB method, or a database query. A transaction T is a tuple $(S, [s, t])$ which represents an instance of execution of a service $S \in \Sigma$ that begins at time s and completes at time t. We call $[s, t]$ an execution period of $T = (S, [s, t])$.

A transaction history D is a historical sequence of transactions obtained by the monitoring instrumentation installed in the computer system. Many instrumentation methods can collect some kind of data that can be modeled as D. We assume that D is sorted in the order of completion time.

If the execution period of a transaction $T_1 = (S_1, [s_1, t_1])$ contains that of another transaction $T_2 = (S_2, [s_2, t_2])$, namely, $s_1 \leq s_2 \leq t_2 \leq t_1$, then we say that T_1 contains T_2 and denote this by $T_1 \succeq T_2$. If a transaction T_1 calls another transaction T_2 then we say that T_1 *directly* depends on T_2, and denote this by $S_1 \Rightarrow S_2$. We also say that T_1 is a parent of T_2 and T_2 is a child of T_1. Note that each transaction has (at most) one parent. If T_1 directly depends on T_2 or T_1 is a parent of another transaction T_3 that depends on T_2 (a recursive definition), we say that T_1 depends on T_2 and denote this by $T_1 \rightarrow T_2$

We say that a service S_1 *directly* depends on another service S_2 and denote this by $S_1 \Rightarrow S_2$ if there exist $T_1 = (S_1, [s_1, t_1])$ and $T_2 = (S_2, [s_2, t_2])$ in D such that T_1 *directly* depends on T_2. Also, we say that a service S_1 depends on S_2 and denote this by $S_1 \rightarrow S_2$ if there exist $T_1 = (S_1, [s_1, t_1])$ and $T_2 = (S_2, [s_2, t_2])$ in D such that T_1 depends on T_2.

A dependency may be *conditional*, that is, even if S_1 depends on S_2, not all transactions of S_1 in D have to depend on a transaction of S_2, and there may be some transaction $T = (S_1, [s, t]) \in D$ such that T does not depend on any transaction of S_2. In addition, a transaction of S_1 may call another service S_2 more than once. We will later define a *strength* of $S_1 \Rightarrow S_2$ or $S_1 \rightarrow S_2$ by using the probability or the number of times a transaction of S_1 calls transactions of S_2. $G(D)$ is a set of all dependencies that can be observed in a given transaction history D. Since a dependency is a binary relationship, G can be represented as a directed graph. We assume that transactions are *synchronous*, that is, if T_1 calls T_2, T_1 must wait until T_2 completes. Thus $T_1 \succeq T_2$ if

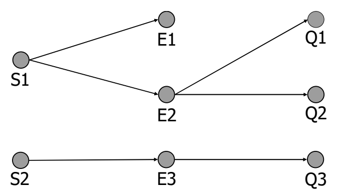

Figure 2. Example of dependency graph G

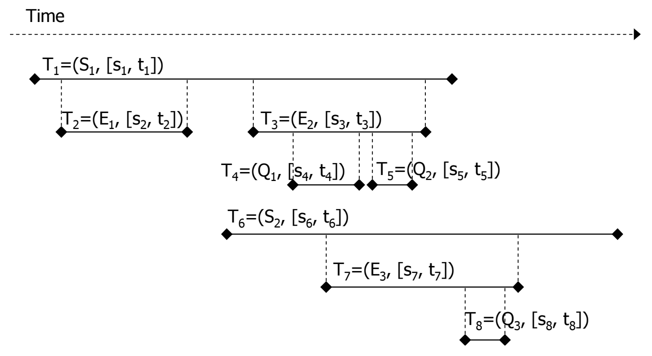

Figure 3. Concurrent execution of S_1 and S_2

$T_1 \rightarrow T_2$. Furthermore, if T_1 calls $T_2 = (S_2, [s_2, t_2])$ and $T_3 = (S_3, [s_3, t_3])$, then the execution period of T_2 and T_3 must not overlap. In other words, T_3 must begin after T_2 completes or T_2 must begin after T_3 completes. Therefore, $[s_2, t_2] \cap [s_3, t_3] = \emptyset$ holds.

Let us review the above notions by using an example shown in Figure 2, in which services are represented as nodes and dependencies are represented as arcs. There are two servlets $\{S_1, S_2\}$, three EJB methods $\{E_1, E_2, E_3\}$, and three database queries $\{Q_1, Q_2, Q_3\}$. Thus $\Sigma = \{S_1, S_2, E_1, E_2, E_3, Q_1, Q_2, Q_3\}$. There are arcs from S_1 to E_1 and E_2, which implies that S_1 *directly* depends on (i.e. calls) E_1 and E_2. On the other hand, there is a path from S_1 to Q_1, but there are no edge from S_1 to Q_1, which implies that S_1 depends on Q_1, but S_1 does not directly depend on Q_1.

Suppose that S_1 and S_2 are invoked by different clients independently and concurrently as shown in Figure 3. A transaction $T_1 = (S_1, [s_1, t_1])$ is an execution of the servlet S_1 which begins at time s_1 and finishes at time t_1. Since S_1 calls E_1 and E_2, T_1 invokes $T_2 = (E_1, [s_2, t_2])$ and $T_3 = (E_2, [s_3, t_3])$. Likewise, T_3 invokes $T_4 = (Q_1, [s_4, t_4])$ and $T_5 = (Q_2, [s_5, t_5])$; and $T_6 = (S_2, [s_6, t_6])$ calls $T_7 = (E_3, [s_7, t_7])$ which calls $T_8 = (Q_3, [s_8, t_8])$. Since transactions are synchronous, $T_1 \succeq T_2, T_3$; $T_3 \succeq T_4, T_5$; and $T_6 \succeq T_7 \succeq T_8$. In this figure $T_6 \succeq T_3, T_4, T_5$ and $T_7 \succeq T_5$ but those just happened by chance.

So far we have considered as if we know which transaction calls which ones. If we know such information, it is straightforward to obtain dependencies among services. For example, ARM [21] is a set of APIs that allows users to capture the behaviors of applications, and we can know true pairs of a parent transaction and a child transaction by using correlation IDs given by ARM. However, ARM loads systems much since the target applications must call ARM APIs several times for each transaction. Also, we need changes in source code of the target applications to enable ARM. Therefore, monitoring such dependencies between individual transactions is too costly and infeasible in most cases. Instead, monitoring execution time period of each transaction, in other words, obtaining a transaction history D is rather easy and practical. In this paper, we take the approach of inferring dependencies from D. In the following sections, we introduce our network-based approach for obtaining D in a distributed environment, and then propose a plausible estimation of dependency relationships $G(D)$.

3 Network-based transaction monitoring

In this section, we discuss how to collect a transaction history D from a running distributed system. In [13], Gupta *et al.* proposed a host-based approach to obtain execution time periods of transactions, in which performance metrics managed by common performance monitoring instrumentation are periodically polled. The proposed method is very effective for discovering dependencies *within a node* (i.e., a server machine). However, to extend the method to discover dependencies *across distributed nodes*, we have to synchronize the clocks of nodes with very high precision and we need to gather data from nodes to a single node to apply a dependency discovery algorithm.

Usually, components in an enterprise system are often installed on distinct nodes, and communication among them is done via network. Therefore we take a network-based approach, in which we monitor network data at a network device to obtain transaction execution time periods. Many modern network devices (routers, switches, or hub) are capable of configuring a mirror port that receives all (or defined part of) network traffic passing through the device with no or little performance overhead. By analyzing captured network traffic, we can restore start and stop times of transactions. For instance, we can extract the start time of an HTTP transaction from a packet including 'GET' method, and the end time from the packet including the message 'HTTP/1.1 200 OK'. Since we can observe data for multiple nodes at a single node attached to a mirror port, no time synchronization is necessary. In addition, a network packet contains the IP address of a node that submits a transaction request, which can be a clue to restrict candidates of parent transactions (i.e., the parent transaction must run on the node). A limitation of this approach is that we may not be able to capture all the data passing through the network de-

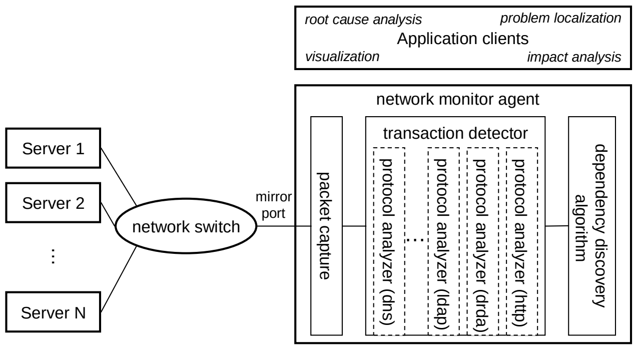

Figure 4. A prototype system

vice when the total traffic volume exceeds the capacity of a mirror port (e.g., 1 Gb/s). Although we have to design dependency discovery algorithms to be tolerant to packet losses, this property can be considered as an adaptive sampling method that regulates the volume of input data for the dependency discovery algorithms.

We have developed a prototype system to demonstrate effectiveness of our approach as illustrated in Figure 4. The current prototype supports only a few protocols including HTTP, DRDA (used by DB2), DNS, and LDAP, but the architecture allows us to add protocol analyzers to extend the system to support other transaction types.

4 An existing method for discovering dependencies

Gupta *et al.* presented in [13] an on-line dependency extraction algorithm (ODEA) that efficiently discovers dependencies (defined below) from a given transaction history. ODEA measures the strength of dependency $S_1 \rightarrow S_2$ by the probability p with which a transaction of S_1 contains at least one transactions of S_2. More formally, for a given transaction history D, the strength p (or p-value) of the dependency $S_1 \rightarrow S_2$ is defined as $p = \sharp(S_1|S_1 \succeq S_2)/\sharp(S_1)$, where $\sharp(S_1)$ is the number of transactions of S_1 in D, and $\sharp(S_1|S_1 \succeq S_2)$ is the number of transactions of S_1 in D that contains at least one transaction of S_2. Note that one transaction of S_1 contributes to $\sharp(S_1|S_1 \succeq S_2)$ by just one even if it contains more than one transactions of S_2. ODEA is an on-line algorithm that efficiently discovers every pair of services (S_1, S_2) such that the p-value of $S_1 \rightarrow S_2$ is non zero (or large). If there is a *true* dependency between S_1 and S_2, that is, S_1 directly or indirectly calls S_2, then there must be transactions of S_1 in D that contain S_2's transactions, and p-value of $S_1 \rightarrow S_2$ is non zero, and hence, ODEA is able to discover all true dependencies. However, even if S_1 does not directly or indirectly call S_2, a transaction of S_1 may contain that of S_2 by chance. Therefore ODEA may discover *false* dependencies. To cope with this problem, [13] introduces another measure r (called r-value) for $S_1 \rightarrow S_2$ which is the probability with which a transaction of S_2 is contained by at least one transactions of S_1, that is defined as

$$r = \frac{\sharp(S_2|S_1 \succeq S_2)}{\sharp(S_2)}, \qquad (1)$$

where $\sharp(S_2|S_1 \succeq S_2)$ is the number of transactions of S_2 that is contained by at least one transaction of S_1. Then [13] uses $\max(p, r) + pr$ as a heuristic to prioritize discovered dependencies. The rationale of this heuristic given in [13] is that the a dependency is more likely to be true if (i) at least one of p and r is high (expressed as the first term) and (ii) both of p and r are high compared to the case when only one of them is high (as the second term).

ODEA is a simple but powerful tool to automatically extract dependencies only from transaction histories. However, we point three drawbacks in the method.

1. The higher the workload intensity becomes, the higher the probability of accidental containments becomes. which results in overestimating dependencies.

2. Direct dependencies and indirect dependencies are not distinguished. This property complicates dependency graphs, which is an obstacle to understand the target system, and makes root cause analysis more difficult.

3. The measure for the strength of dependency is designed in an ad hoc manner. We want more quantitative measures for dependencies.

In the next section, we introduce a novel method that mitigates these problems.

5 An accurate direct dependency estimation method

5.1 Estimating false dependencies

Let us start with the first problem that we mentioned in the last section. Suppose that we have two services S_1 and S_2 that do not depend on each other at all. In other words, the probability with which a transaction of S_1 calls at least one transactions of S_2 is zero. Similarly, the probability with which a transaction of S_2 is called by any transaction of S_1 is zero.

As workload intensity becomes higher, the probability that a transaction of S_1 contains at least one transactions of S_2 by chance and the probability that a transaction of S_2 is contained by any transaction of S_1 by chance become larger, which leads to more false positive dependencies.

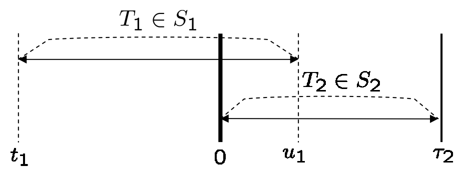

Figure 5. Estimating the probability of accidental containments

For compensating them, we first try to estimate the probabilities of such accidental containments.

Let us assume that requests for S_i arrive randomly according to a Poisson process with workload intensity λ_i ($i = 1, 2$). Then the time interval X_i between start times of two consecutive transactions of S_i follows an exponential distribution with mean $E[X_i] = 1/\lambda_i$ and cumulative probability function $F_i(t) = Pr[X_i \leq t] = 1 - e^{-\lambda_i t}$. Also, let us assume that τ_i follows an exponential distribution with mean $E[\tau_i] = 1/\mu_i$ and cumulative probability function $T_i(t) = Pr[\tau_i \leq t] = 1-e^{-\mu_i t}$ for $i = 1, 2$. These assumptions are very common in standard performance analyzes [3].

Under these assumptions, we estimate $\psi(S_1, S_2)$, the probability with which a transaction T_2 of S_2 is contained by at least one transactions of S_1 by chance.[1]

Let the start time of T_2 be $t = 0$, and the number of transactions of S_1 that are being executed at time $t = 0$ be N. Then, the probability distribution of N becomes a Poisson distribution $Po(\lambda_1/\mu_1)$,

$$Pr[N = n] = \frac{\lambda_1^n}{\mu_1^n n!} e^{-\frac{\lambda_1}{\mu_1}}.$$

Suppose that there are n transactions of S_1, and let the start time of one of the transactions be t_1, and its end time be u_1 (see Figure 5). Then, u_1 follows an exponential distribution with mean $1/\mu_1$, and the probability of T_1 not containing T_2 is written as

$$fail(\tau_2) = \int_0^{\tau_2} \mu_1 e^{-\mu_1 u_1} du_1 = 1 - e^{-\mu_1 \tau_2}.$$

Therefore, since the end times of the n transactions are independent of each other, the probability with which n transactions of S_1 exist and all of them do not contain T_2 is

$$x_n = \int_0^{\infty} \frac{\lambda_1^n}{\mu_1^n n!} e^{-\frac{\lambda_1}{\mu_1}} fail(\tau_2)^n \mu_2 e^{-\mu_2 \tau_2} d\tau_2.$$

[1]We can also estimate the probability with which a transaction T_1 of S_1 contains at least one transactions of S_2 by chance, but we omit its derivation since it is not used in the following arguments.

Finally, we obtain

$$\psi(S_1, S_2) = 1 - \sum_{n=0}^{\infty} x_n, \qquad (2)$$

where

$$x_n = \frac{\lambda_1^n e^{-\frac{\lambda_1}{\mu_1}}}{\prod_{k=1}^n (k\mu_1 + \mu_2)} = \frac{\lambda_1}{n\mu_1 + \mu_2} x_{n-1}.$$

$\psi(S_1, S_2)$ has an infinite sum, but it is enough to evaluate the sum up to moderate n in practical use.

5.2 Competitive modeling for discovering direct dependencies

In this subsection, we tackle the second problem of the last section by using the result of the previous subsection. The reason why indirect dependencies are extracted is that the previous method [13] does not consider an important constraint that a transaction has at most one parent.

Therefore, we propose a competitive model that takes into account of the constraint explicitly, which is expressed as

$$\sum_i \rho(S_i, S_j) = 1 \quad (0 \leq \rho(S_i, S_j) \leq 1), \qquad (3)$$

where $\rho(S_i, S_j)$ is the probability that a transaction of S_j is directly depended on by a transaction of S_j.[2]

Given a transaction history D, we estimate this model by maximum likelihood approach. Let us first consider by using an example. Suppose that there are four services $\Sigma = \{S_1, S_2, S_3, S_4\}$, and that a transaction T_1 of S_1 is contained by at least one transaction of each of S_2 and S_3, and is not contained by any transaction of S_4. Then the probability that this situation happens is the sum of the probabilities of the following two possible cases; (i) T is directly called by a transaction of S_2, and is contained by transactions of S_3 by chance, and is not contained by any transactions of S_4 by chance; (ii) T is directly called by a transaction of S_3, and is contained by transactions of S_2 by chance, and is not contained by any transactions of S_4 by chance. Therefore, the probability is $\rho(S_2, S_1)\psi(S_3, S_1)(1 - \psi(S_4, S_1)) + \psi(S_2, S_1)\rho(S_3, S_1)(1 - \psi(S_4, S_1))$, where $\psi(S_i, S_j)$ is the probability of a transaction T of service S_j is contained by transactions of service S_i by chance.

Generally, the sum of the log-likelihood for a given transaction history D is

$$L = \sum_{T \in D} \log \sum_{S \in C(T)} \nu(S, C(T)|T), \qquad (4)$$

[2]To incorporate services depended on by no other services, we suppose a virtual transaction $T_0 = (S_0, [-\infty, \infty])$ of a virtual service S_0.

where $C(T) \subseteq \Sigma$ is the set of services of transactions by which a transaction T is contained,

$$\nu(S, C(T)|T) = \rho(S, S(T)) \prod_{S' \in C(T) \setminus S} \psi(S', S(T)) \cdot \prod_{S'' \notin C(T)} (1 - \psi(S'', S(T))),$$

and $S(T)$ is the service of a transaction T.

Our goal is to compute $\rho(S_i, S_j)$ that maximizes the likelihood L subject to the constraint (3). Since it is difficult to solve this optimization problem analytically, we propose an iterative method based on the Expectation Maximization (EM) algorithm [9] to guarantee that each iteration makes the likelihood better. Since L has no local maxima, it is guaranteed that we can always find the optimal solution.

Theorem: The sum of the log-likelihood (4) for the transaction history D has no local maxima.

Proof: The Hessian of L is negative semi-definite. See Appendix A for details. $\Box$

The parameter estimation is performed as follows. In the expectation step, based on the current ρ, the probability that transaction T is directly called by transactions of service S_i is computed by

$$\begin{aligned}\Pr(S_i|C(T), T) &= \Pr(S_i, C(T)|T)/\Pr(C(T)|T) \\ &= \frac{\delta(S_i \in C(T))\rho(S_i, S(T)) \prod_{S' \in C(T) \setminus S_i} \psi(S', S(T))}{\sum_{S \in C(T)} \rho(S, S_j) \prod_{S' \in C(T) \setminus S} \psi(S', S(T))},\end{aligned} \quad (5)$$

where δ is a function that returns 1 when its argument is true, and returns 0 otherwise. Therefore, *service call frequency* $\hat{\sharp}(S_i \Rightarrow S_j)$, the expected number of times transactions of S_i called transactions of S_j, is obtained by

$$\hat{\sharp}(S_i \Rightarrow S_j) = \sum_{T \in D(S_j)} \Pr(S_i|C(T), T). \quad (6)$$

In the maximization step, the next maximum likelihood estimation of the parameters are computed by

$$\rho(S_i, S_j) = \frac{\hat{\sharp}(S_i \Rightarrow S_j)}{\sharp(S_j)}. \quad (7)$$

Getting these steps together, the update rule for $\rho(S_i, S_j)$ becomes

$$\rho(S_i, S_j) \leftarrow \rho(S_i, S_j) \frac{1}{\sharp(S_j)} \frac{\partial L}{\partial \rho(S_i, S_j)}, \quad (8)$$

since

$$\frac{\partial L}{\partial \rho(S_i, S_j)} = \sum_{T \in D(S_j)} \frac{\delta(S_i \in C(T)) \prod_{S' \in C(T) \setminus S_i} \psi(S', S_j)}{\sum_{S \in C(T)} \rho(S, S_j) \prod_{S' \in C(T) \setminus S} \psi(S', S_j)}.$$

Therefore, starting at any initial parameters, optimal parameters are obtained by applying (6) and (7) alternatively until convergence.

Although this update assumes a batch processing of the whole D, we can also use the on-line EM algorithm [22] that process one transaction at each update. Receiving a transaction T, the parent probabilities (5) for T are calculated based on the current ρ, and then added to (6) to update the parameters.

In contrast to ODEA considering only whether S_i depends S_j or not, service call frequency $\hat{\sharp}(S_i \Rightarrow S_j)$ indicates the estimated number of calls, and is a quantitative measure of dependency for understanding the behavior of the system more deeply. We define the service call frequency matrix F as $f_{ij} = \hat{\sharp}(S_i \Rightarrow S_j)$.

To avoid the unwanted effects of fluctuation of workload intensity, a normalized quantity $h_{ij}(f_{ij})$ is useful for problem detection instead of f_{ij} itself, where $h_{ij}(\cdot)$ is a monotonic scaling function. One reasonable way is to define the *service call ratio* (SCR) matrix C by choosing $h_{ij}(\cdot) = 1/\sharp(S_i)$. The (i, j) element of C is given by

$$c_{ij} = \frac{\hat{\sharp}(S_i \Rightarrow S_j)}{\sharp(S_i)}. \quad (9)$$

Namely, this is the expected number of times that *each* transaction of service S_i directly calls service S_j. When the target system is stable, c_{ij} is expected to show a constant value regardless of workload intensity and can be a powerful tool for root cause analysis. Another way is to define a *scaled service call frequency matrix* K by choosing $h_{ij}(\cdot) = \ln(1 + \cdot)$. Empirically, the logarithmic transformation is known to be effective in eliminating the bursty nature of traffic. Note that this scaling function does not depends on (i, j), so that the balance between services is conserved. This property is preferable to global anomaly metric watching the overall state of the system.

Based on these metrics, C and K, we propose an approach for problem detection in the next section.

6 Problem detection

6.1 Individual dependencies

To pursue a reasonable problem detection method, let us take a look at the characteristics of dependency matrix. We conducted a preliminary experiment under the condition described in Section 7. By using the algorithm introduced in the previous section, we calculated F when the system exhibited no failures. Figure 6 shows an example of the time dependence of F. We updated the matrix every 5 minutes. While the generated matrix was 23×23 square matrix, only seven dependencies out of 253 are plotted. They are such

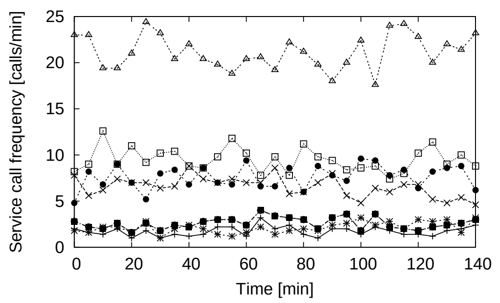

Figure 6. The number of calls from HTTP Server to application server.

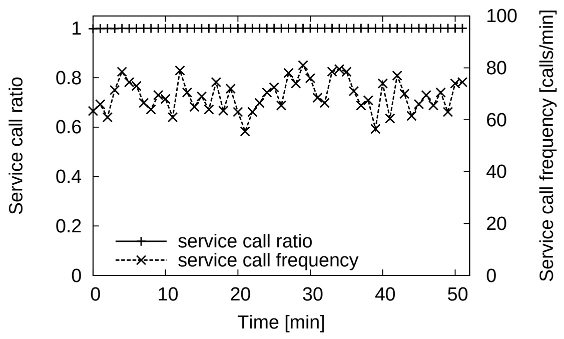

Figure 7. Comparison between service call frequency and service call ratio.

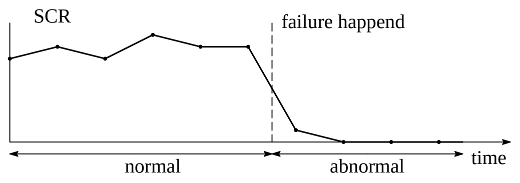

Figure 8. Anomaly detection based on Criterion 1.

dependencies that related to the communication between an HTTP server and an application server. From the figure, we can see that the matrix elements f_{ij} are quite heterogeneous and greatly vary over time. Therefore the number of calls itself is not useful for diagnosing the system.

Then we show the time development of the SCR matrix C in Figure 7, comparing with that of F for the same dependency corresponding to a communication between the HTTP and application servers. Since clients randomly call the HTTP server, the service call frequencies considerably vary over time. In contrast, the SCR in this case is relatively stable because HTTP servers are designed to call application servers with a constant number per a client request. In this case, the constant number is one.

This observation leads us to a simple rule to pick up anomalies:

Criterion 1: A service dependency is anomalous if the service suddenly gets inactive, where active services are defined as those maintain SCR values larger than a threshold value over a predetermined consecutive period.

We depicted this criterion in Figure 8. For the threshold, the value of zero is generally a practical choice since tuning a threshold value is often a tough task unless one has a detailed model of the system. While simple, this criterion is useful in many cases and provides a unique feature to our network-based monitoring system. In fact, a considerable amount of application faults occurs at a single service and can be detected by watching the corresponding dependency irrespective of the complexity of service dependency graph.

However, there is a class of failures that cannot be captured by Criterion 1. First, some of the SCR dependencies are unstable and do not take constant values as in Figure 7, so that the simple thresholding criteria do not work well for such dependencies. For example, some SCRs directly depend on parameters randomly chosen by clients, resulting in unstable SCR values. Second, as indicated in Figure 6, some service dependencies may have correlation with others. This fact suggests another type of failures which appears themselves in the relationship between services, maintaining finite values of dependencies. Similarly, one cannot know how large an observed amount of fluctuation is through watching a single dependency. The intensity of fluctuation should be compared with the average scale of fluctuations in the whole system.

Considering these issues, we propose a hierarchical scenario of problem detection as follows (see Figure 9). First, we compute an anomaly metric that captures the whole system based on an unsupervised learning framework. Once an anomaly is detected, we step down to a lower layer to examine each service using a properly defined metric. If faulty services are identified, we further step down to the third layer, where SCRs are utilized to localize the fault. We discuss the first and second layers in the subsequent subsections.

6.2 Service activities

To extract a feature for individual services, we employ the framework of Idé-Kashima [17]. We recapitulate the essence below. First, we define the feature vector $\boldsymbol{u}$ of the dependency matrix as

$$\boldsymbol{u}(t) \equiv \arg\max_{\tilde{\boldsymbol{u}}} \left\{ \tilde{\boldsymbol{u}}^T \tilde{\mathsf{K}}(t) \tilde{\boldsymbol{u}} \right\} \tag{10}$$

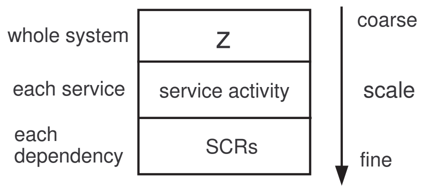

Figure 9. Hierarchical view of problem detection.

subject to $\tilde{\boldsymbol{u}}^T \tilde{\boldsymbol{u}} = 1$, where T denotes transpose. Here we used a symmetrized version of the scaled frequency matrix, $\tilde{\mathsf{K}}$, whose (i, j) element is defined as $k_{ij} + k_{ji}$ in order to maintain the consistency as a dynamical system [17]. Since $\tilde{\mathsf{K}}$ is a non-negative matrix, one can see that the maximum value is attained if the weight of $\boldsymbol{u}(t)$ is larger for services where the (i, j) element of $\tilde{\mathsf{K}}$ is larger. If a service i actively calls other services, $\boldsymbol{u}(t)$ has a large weight for the i-th element. Following this interpretation, we call this feature vector an *activity vector* [17].

One important experimental fact is that strong random fluctuations in F are considerably suppressed by putting together the information in the matrix into $\boldsymbol{u}$. Encouraged by this fact, we consider a simple probabilistic model for individual service activities. We employ a Gaussian model where each activity independently fluctuates. Generally, the correlation between elements should be taken into account, but the SCR matrix well complements the information.

For each service i, the average $w_i(t)$ and the standard deviation $\sigma_i(t)$ at time t can be updated using well-known on-line maximum likelihood formula [26] such as

$$w_i(t) = (1 - \beta) \cdot w_i(t-1) + \beta u_i(t), \tag{11}$$

where β is the discounting factor. Now we have the second criterion for problem detection.
Criterion 2: A service is faulty if

$$|u_i(t) - w_i(t)| > \sigma_i(t) x_{\text{th}}$$

holds, where x_{th} is the solution of

$$\int_{x_{\text{th}}}^{\infty} du N(x) = p_c.$$

Here, p_c is the critical boundary (< 1) and $N(x)$ represents the standard normal distribution.

6.3 Overall state

Next, for a time sequence of the activity vector, we introduce an anomaly metric $z(t)$, defined as

$$z(t) \equiv 1 - \boldsymbol{r}(t)^T \boldsymbol{u}(t), \tag{12}$$

where $\boldsymbol{r}(t)$ is the typical activity pattern defined at t. The value of $z(t)$ is unity if the present activity vector is orthogonal to the typical pattern at t, and zero if the present activity vector is identical to the typical pattern.

To find $\boldsymbol{r}(t)$, let $\mathsf{U}(t)$ be

$$\mathsf{U}(t) = [\boldsymbol{u}(t-1), \boldsymbol{u}(t-2), \cdots, \boldsymbol{u}(t-W)], \tag{13}$$

where W is a window size, and define $\boldsymbol{r}(t)$ as the principal left singular vector of $\mathsf{U}(t)$, where a singular vector is said to be principal if it corresponds to the largest singular value.

To relate the value of z with a universal quantity, we consider a generative model for $\boldsymbol{u}$. Since $\boldsymbol{u}$ is normalized, a general assumption is that the PDF of $\boldsymbol{u}$ can be modeled as the von Mises-Fisher (vMF) distribution. Idé-Kashima [17] showed that a generative model with respect to z can be derived from the vMF distribution as

$$q(z|n, \Sigma) = \frac{1}{(2\Sigma)^{\frac{n-1}{2}} \Gamma(\frac{n-1}{2})} e^{-z/(2\Sigma)} z^{\frac{n-1}{2}-1}, \tag{14}$$

where n is the effective dimension of the system. The parameters of Σ and n in this model can be exactly evaluated using the first and second moments as

$$n = 1 + \frac{2{m_1}^2}{m_2 - {m_1}^2} \quad \text{and} \quad \Sigma = \frac{m_2 - {m_1}^2}{2m_1}, \tag{15}$$

where m_k is defined as $\int q(z|n, \Sigma) z^k dz$ for $k = 1, 2$. The moments are easily calculated on-line in a similar manner to Eq. (11). Now, we have the third criterion:
Criterion 3: The system is anomalous if

$$z(t) > z_{\text{th}}(t)$$

holds, where $z_{\text{th}}(t)$ is the solution of

$$\int_{z_{\text{th}}}^{\infty} dz\, q(z|n, \Sigma) = p_c.$$

The parameters n and Σ are evaluated at t.

7 Experiments

In this section, we validate our approach to problem detection in a benchmark system. Our experiment consists of two parts. Firstly, we evaluate the accuracy of our method of direct dependencies, and secondly, we demonstrate the utility of our problem detection framework based on the extracted dependency information.

7.1 Experimental settings

We built a three-tier Web-based system on Linux (2.4.18-14arp) using IBM HTTP Server 1.3.26, IBM WebSphere

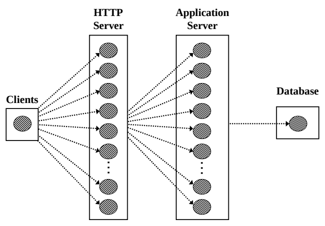

Figure 10. Target 3-tier system and services defined in Trade3 application.

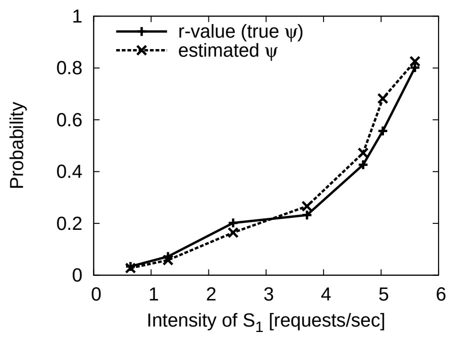

Figure 11. Estimated probabilities of accidental containments

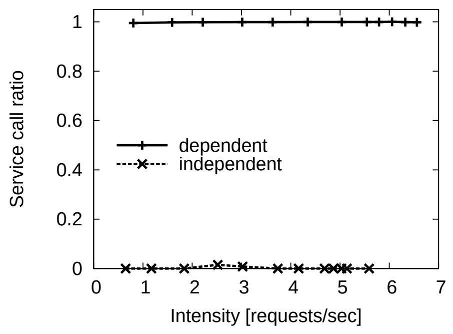

Figure 12. SCRs between two services

Application Server 5.0.2, and IBM DB2 Universal Database Enterprise Server Edition Version 8.1, as shown in Figure 10.

An application called “Trade3” is running on this system. Trade3 is one of the standard benchmark applications for end-to-end performance of WebSphere [16], and models an on-line stock brokerage activity. Receiving a request from a client, the HTTP server forwards it to the application server. Servlets and JSPs running on the application server handle it and invoke methods on EJBs. Invoked EJBs interact with the database server by making JDBC calls. The application takes one of 10 values as a parameter: ‘login’, ‘home’, ‘account’, ‘update profile’, ‘quote’, ‘portfolio’, ‘buy’, ‘sell’, ‘register’ and ‘logout’. Each value corresponds to an action that a client takes. For example, a client can perform an operation such as login, buy a stock, get quotes and view portfolio by sending an HTTP request with one of the parameters to the HTTP server.

We used the transaction monitor described in Section 3 to extract transactions from network data. A service is defined as a 4-tuple (“IP address”, “TCP port number”, “protocol”, “parameter”). For instance, (ip_1, 80, HTTP, /trade/app?action=login) is a service. Running this application on our system, the total number of observed services amounted to 23.

In all the experiments, we generated requests to the HTTP server using a workload generator that simulates multiple virtual users concurrently accessing the Web application. We control the workload intensity by changing the number of virtual users.

7.2 Direct dependency discovery

Before proceeding to problem detection, we assess the quality of discovered dependencies used in problem detection.

First, we take independent service pairs and compare estimated accidental containment probability ψ defined in Eq.(2) with observed containment probability r defined in Eq.(1). Note that the r-value is equal to the *true* accidental containment probability for independent services. Figure 11 shows the result of a case when S_1=(ip_1, 80, HTTP, /trade/app?action=portfolio) and S_2=(ip_2, 9081, HTTP, /trade/app?action=home), which are known to be independent of each other. We can see the probability of accidental containments (i.e., false positive dependency) rapidly increases as the workload intensity increases. Our estimation fits well even when the workload intensity is high.

Next, we investigate the accuracy of SCR estimated by combining the competitive model with the estimated accidental containment probability. In Figure 12, the line labeled ‘independent’ indicates the SCR between S_1 and S_2. The true value of the SCR is 0 because they are independent, and we can see that our estimation fits it very well. The line labeled ‘dependent’ in the figure shows the SCR between S_1'=(ip_1, 80, HTTP, /trade/app?action=home) and S_2. Since S_1' directly calls S_2 once when S_1' is called, the true SCR for $S_1' \Rightarrow S_2$ is 1, and again, the estimation fits it very well.

Although we showed only two examples here, the same trend was observed for the other pairs of services.

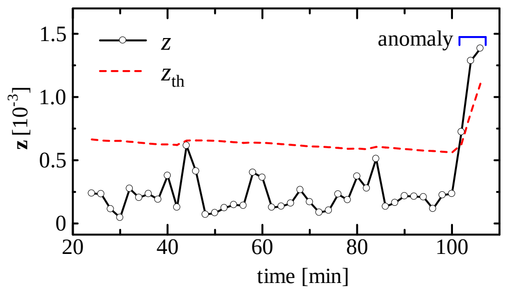

Figure 13. Time dependence of the anomaly metric and its 0.5% threshold.

7.3 Problem detection and localization

To validate the ability of anomaly detection, we injected an artificial fault in Trade3 application so that a particular servlet starts to encounter a runtime exception at a specified time. The servlet catches the exception and returns a corrupted HTTP page to the client. This type of failures are common and hard to detect using existing techniques that watch the return codes of service calls and event logs recorded by servers, because each service execution returns without error, and we see too many runtime exceptions in a log file even in a normal state. Therefore system administrators often become aware of such failures by receiving complaints from the client users. We expect that the proposed method can detect this kind of failures well, since the servlet changes its internal execution path to handle the exception, which will affect the frequency of subsequent service invocations.

We conducted the on-line problem detection task based on the scenario shown in Figure 9. We collected data at the workload intensity of 8.28 requests for the HTTP server per second, and the fault was injected at $t = 104$ minutes. We generated C and F every 2 minutes, where the accidental containment probability ψ (2) was estimated by the data for recent 2 minutes.

First, we computed z and its threshold value on-line. For parameters, we took $W = 10$, $\beta = 0.025$, and $p_c = 0.5\%$. The result is shown in Figure 13. We see that the value of z exceeds $z_{\rm th}$ at the last three time points, indicating an anomaly. We also see that $z_{\rm th}$ increases after experiencing the high z values because of on-line learning.

To identify faulty services, we examined the service activities using Criteria 2. Figure 14 shows the ratio

$$\gamma_i \equiv \frac{|u_i(t) - \mu_i|}{\sigma_i(t) x_{\rm th}}$$

at $t = 104$ minutes for each service. If either upper or lower threshold is exceeded, $\gamma_i > 1$, otherwise $0 < \gamma_i <$

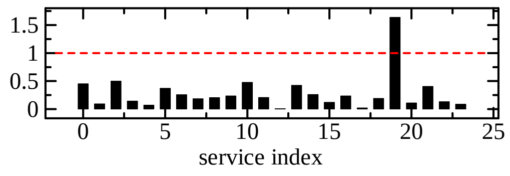

Figure 14. The γ value at $t = 104$.

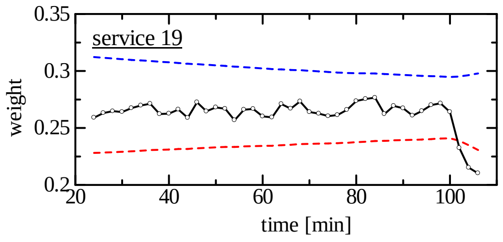

Figure 15. Time dependence of the activity of the service 19.

1. As clearly shown, the activity of the service 19 exhibits an anomaly. The time dependence of the activity of this service is shown in Figure 15, together with upper and lower thresholds of $p_c = 0.5\%$ (dashed curves). We used the same discounting factor as above. We see that $u_{19}(104)$ clearly exceeds the lower threshold when $t > 102$ minutes. We also see that the activity is relatively stable during the normal state of the system in spite of the bursty nature of traffic. This result well validates our assumption of the individual Gaussian model.

Finally, we referred to the SCR matrix to identify faulty components. This anomalous service was S_2 defined in the previous subsection. Comparing the SCR matrices at before and after the malfunction and following edges with larger weight in the dependency graph, it was evident that the dependency on the service of (ip_3, 50000, DRDA, TRADE3DB) indicates the root cause of the problem because the SCR value has suddenly decreased from 16.1 to 1. This means the service S_2 was being executed very little DB calls with the parameter of "/trade/app?action=home" due to a fault.

Note that this type of inference could get much difficult if the service call frequency matrix itself were used since each value of the elements simply represents the number of calls, which is strongly traffic dependent. In addition, the same holds if the dependency matrix included *in*direct dependencies. From a viewpoint of graphical models [19], our dependency estimation method corresponds to graphical modeling with only two-way interactions. We will discuss the detail of inference techniques using the SCR dependency graph in other publications.

8 Related work

Dependency information are categorized into two groups, static dependencies and dynamic dependencies. Static dependency discovery relies on a human or static analysis program to analyze system configuration, installation data, and application code. Kar *et al.* described a method for automatically extracting static dependencies between software components within a given machine from existing software deployment repositories in [18].

However, this method is not capable of obtaining dynamic dependencies that are established during the runtime operation of the system, where many components dynamically bind with each other at runtime, and the dependencies often change over time as the environment changes. Dynamic dependency discovery operates at runtime and collect information from the system to derive dynamic dependencies. There are several methods that use special instrumentation to capture dynamic dependencies from the system [4, 21, 7]. Sometimes, it is desirable to infer dependencies from data that can be obtained more easily without loading systems much [13, 10]. Ensel used Neural Networks to automatically generate dynamic dependencies by looking at pairs of systems behavior such as CPU load over time [10]. Our approach is also categorized into this group.

In the literature, there are many studies on diagnosis and root cause analysis using dependency information. Event correlation systems (e.g., [27, 8, 12]) collect alarms or events, map them onto corresponding nodes of the dependency graph, and then the dependencies from those nodes are examined to identify the set of nodes on which the most nodes having alarms depend. Another technique for using dependency information for root-cause analysis is to make use of the dependency graph as a guide for systematic examination of the system with a set of probes or test transactions (e.g., [11, 6]).

Hellerstein *et al.* discussed in [15] how data mining techniques can be used to identify actionable patterns from historical event data collected in computer systems. Ma *et al.* addressed the system aspect of event browsing and event mining in [20]. The problem we addressed in this paper can be considered as a variant of association rules discovery [1] or sequential pattern discovery [2]. In this paper, we focused on pair-wise dependencies only, the idea may be naturally generalized to consider dependencies among more than two service.

Recent studies on signal processing have demonstrated the utility of change-detection techniques to characterize anomalies of network traffic [5, 24]. However, most of those studies address highly aggregated data at the lower layers. We focus on the problem at the application layer in Web-based systems, where traditional autoregressive models with white noise are not appropriate because of the strong fluctuation and the heavy tail nature of data. Hajji [14] applied a generative model approach of Yamanishi *et al.* [26, 25] to a fault detection task in a local-area network. However, it only uses information from a single observation point, and considers only lower layer quantities. Thottan-Ji [23] proposes an interesting method to correlate multiple observation points. However, it is based on the autoregressive model and its thresholding policy is not probabilistically consistent. Overall, our contribution is to discuss the anomaly detection task in terms of a vector space model, and to give a probabilistically consistent anomaly detection method.

9 Conclusion

In this paper, we considered automatic detection of problems in distributed systems. For this goal, we proposed (i) a network-based method that obtains the execution time periods from network data without any performance impact; (ii) a new data mining method for discovering direct dependencies among services in distributed systems from historical execution time periods; and (iii) a hierarchical problem detection framework using the discovered dependencies.

We also demonstrated that our approach is promising for problem detection by using a prototype system.

References

[1] R. Agrawal, T. Imielinski, and A. N. Swami. Mining association rules between sets of items in large databases. In *Proceedings of the 1993 ACM SIGMOD International Conference on Management of Data*, pages 207–216. ACM Press, 1993.

[2] R. Agrawal and R. Srikant. Mining sequential patterns. In *Proceedings of the 11th International Conference on Data Engineering*, pages 3–14. IEEE Computer Society, 1995.

[3] A. O. Allen. *Probability, Statistics, and Queuing Theory with Computer Science Applications*. Computer Science and Scientific Computing. Academic Press, 1990.

[4] J. Aman, C. K. Eilert, D. Emmes, P. Yocom, and D. Dillenberger. Adaptive algorithms for managing a distributed data processing workload. *IBM Systems Journal*, 36(2):242–283, 1997.

[5] P. Barford, J. Kline, D. Plonka, and A. Ron. A signal analysis of network traffic anomalies. In *Proceedings of the Second ACM SIGCOMM Workshop on Internet Measurment*, pages 71–82, 2002.

[6] M. Brodie, I. Rish, and S. Ma. Optimizing probe selection for fault localization. In *Proceedings of the 12th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM01)*, 2001.

[7] M. Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer. Pinpoint: Problem determination in large, dynamic internet

services. In *Proceedings of the International Conference on Dependable Systems and Networks*, 2002.

[8] J. Choi, M. Choi, and S. Lee. An alarm correlation and fault identification schema based on OSI managed object classes. In *Proceedings of the IEEE International Conference on Communications*, pages 1547–1551, 1999.

[9] A. P. Dempster, N. M. Laird, and D. B. Rubin. Maximum likelihood from incomplete data via EM algorithm. *Journal of the Royal Statistical Society*, B(39):1–38, 1977.

[10] C. Ensel. New approach for automated generation of service dependency models. In *2nd Latin American Network Operation and Management Symposium, LANOMS*, 2001.

[11] J. Gao, G. Kar, and P. Kermani. Approaches to building self healing systems using dependency analysis. In *Proceedings of IEEE/IFIP Network Operations and Management Symposium (NOMS)*, April 2004. to appear.

[12] B. Gruschke. Integrated event management: Event correlation using dependency graphs. In *Proceedings of the 9th IFIP/IEEE International Workshop on Distributed Systems Operation and Management*, 1998.

[13] M. Gupta, A. Neogi, M. K. Agarwal, and G. Kar. Discovering dynamic dependencies in enterprise environments for problem determination. In *Proceedings of the 14th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management*, volume 2867 of *Lecture Notes in Computer Science*, pages 221–232. Springer, October 2003.

[14] H. Hajji. Baselining network traffic and online faults detection. In *Proceedings of IEEE International Conference on Communications*, volume 1, pages 301–308, 2003.

[15] J. L. Hellerstein and S. Ma. Mining event data for actionable patterns. In *Proceedings of the 26th International Computer Measurement Group Conference*, pages 307–318, December 2000.

[16] IBM. Trade3; `http://www-306.ibm.com/software/webservers/appserv/benchmark3.html`.

[17] T. Idé and H. Kashima. Eigenspace-based anomaly detection in computer systems. In *Proceedings of the Tenth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, 2004.

[18] G. Kar, A. Keller, and S. Calo. Managing application services over service provider networks: Architecture and dependency analysis. In *Proceedings of the 5th IFIP/IEEE International Symposium on Integrated Network Management (IM V)*, 2000.

[19] S. L. Lauritzen. *Graphical Models*. Oxford, 1996.

[20] S. Ma, C. Perng, and J. L. Hellerstein. Eventminer: An integrated mining tool for scalable analysis of event data. In *Proceedings of the Visual Data Mining Workshop*, pages 1–9, August 2001.

[21] OpenGroup Technical Standard C807. Systems management: Application response measurement, 1998.

[22] M. Sato and S. Ishii. On-line EM algorithm for the normalized gaussian network. *Neural Computation*, 12(2):407–432, 2000.

[23] M. Thottan and C. Ji. Anomaly detection in IP networks. *IEEE Transactions on Signal Processing*, 51(8):2191– 2204, 2003.

[24] H. Wang, D. Zhang, and K. G.Shin. Detecting SYN flooding attacks. In *Proceedings IEEE INFOCOM 2002*, pages 1530 –1539, 2002.

[25] K. Yamanishi and J. Takeuchi. A unifying framework for detecting outliers and change points from non-stationary time series data. In *Proceedings of the Eighth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 676–681, 2002.

[26] K. Yamanishi, J. Takeuchi, G. Williams, and P. Milne. On-line unsupervised outlier detection using finite mixtures with discounting learning algorithms. In *Proceedings of the Sixth ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, pages 320–324, 2000.

[27] S. A. Yemini, S. Kliger, E. Mozes, Y. Yemini, and D. Ohsie. High speed and robust event correlation. *IEEE Communications Magazine*, 34(5):82–90, May 1996.

A Proof of Theorem

We prove that L has no local maxima by proving that H, the Hessian of L, is negative semi-definite. H is written as

$$\begin{aligned}[H]_{(i,j),(k,l)} &= \frac{\partial^2 L}{\partial\rho(S_i,S_j)\partial\rho(S_k,S_l)} \\ &= -\sum_{T\in D(S_j)} \delta(S_i\in P(T))\delta(S_k\in P(T)) \\ &\quad \cdot \frac{\prod_{S'\in P(T)\setminus S_i}\psi(S',S_m)\prod_{S''\in P(T)\setminus S_k}\psi(S'',S_m)}{\left(\sum_{S\in P(T)}\rho(S,S_m)\prod_{S'\in P(T)\setminus S}\psi(S',S_m)\right)^2}\end{aligned}$$

for $j = l := m$, and $[H]_{(i,j),(k,l)} = 0$, otherwise. Therefore, H is negative semi-definite if $H_m(T)$ is positive semi-definite for $\forall m = 1, 2, \ldots, |\Sigma|$ and $\forall T \in D$, where

$$\begin{aligned}[H_m(T)]_{j,l} &:= \delta(S_i\in P(T))\delta(S_k\in P(T)) \\ &\quad \cdot \prod_{S'\in P(T)\setminus S_i}\psi(S',S_m)\prod_{S''\in P(T)\setminus S_k}\psi(S'',S_m).\end{aligned}$$

$H_m(T)$ can be decomposed as

$$\begin{aligned}H_m(T) &= \mathbf{h}_m(T)\mathbf{h}_m^\top(T), \\ \mathbf{h}_m(T) &:= (h_m^1(T), h_m^2(T), \ldots, h_m^{|\Sigma|}(T))^\top, \\ h_m^i(T) &:= \delta(S_i\in P(T))\prod_{S'\in P(T)\setminus S_i}\psi(S',S_m).\end{aligned}$$

Therefore $H_m(T)$ is positive semi-definite since

$$\mathbf{x}^\top H_m(T)\mathbf{x} = (\mathbf{x}^\top\mathbf{h}_m(T))^2 \geq 0$$

for $\forall\mathbf{x} \in R^{|\Sigma|}$.