

# Approaches to Building Self Healing Systems using Dependency Analysis

*J. Gao*  
Department of Computer Science  
Stanford University  
Stanford, CA 94305  
USA  
jgao@cs.stanford.edu

*G. Kar, P. Kermani*  
IBM T. J. Watson Research Center  
Hawthorne, NY 10532  
USA  
{gkar, parviz}@us.ibm.com

## Abstract

Typical distributed transaction environments are a heterogeneous collection of hardware and software resources. An example of such an environment is an electronic store front where users can launch a number of different transactions to complete one or more interactions with the system. One of the challenges in managing such an environment is to figure out the root cause of a performance or throughput problem that manifests itself at a user access point and take appropriate action, preferably in an automated way. Our paper addresses this problem by analyzing the dependency relationship among various software components. We also provide theoretical insight into how a set of transactions can be generated to pinpoint the root cause of a performance problem that is manifested at the user access point.

## Keywords

Self healing, problem determination, dependency analysis, transaction performance management, root cause analysis

## 1 Introduction

Typical e-business environments, such as distributed transaction processing systems, are a collection of heterogeneous hardware and software components that interact in very complex ways to support end user transactions. Customers of such systems expect high availability, rapid response time and guaranteed throughput. Such customer expectations are usually captured in Service Level Agreements (SLA) with the provider. When a situation arises such that one or more elements of the SLA is violated, for example a user transaction experiences degraded response time, the root cause of the problem needs to be found rapidly and corrective actions need to be taken to minimize the impact of the fault. This paper details an approach for doing this based on dependency analysis. This work builds on previous effort [10][4]. Consider any two resources, say *A* and *B*, which are part of a distributed system. "*A*" might, for example, be a servlet within a web application server, which implements part of the logic for a business transaction. "*B*" might be an SQL processing agent, such as an EJB, which provides database access to servlet "*A*" for completing the business transaction. These resources are typically monitored by management agents

[19][20], which supply information about their status through a set of observable metrics. In the general case,  $A$  is said to be dependent on  $B$ , if  $B$ 's services are required for  $A$  to complete its own service. We may represent this fact by directed graph with  $A$  and  $B$  as nodes and an edge drawn from  $A$  to  $B$ . A weight may also be attached to the directed edge from  $A$  to  $B$ , which may be interpreted in various ways, such as a quantitative measure for the extent to which  $A$  depends on  $B$  or how much  $A$  may be affected by the non-availability or poor performance of  $B$ , etc. In this paper we are interested primarily in designing and analyzing algorithms for managing end-user transactions and, therefore, are interested in their dependency on all, i.e. the complete set of monitored resources in the distributed system that supports these transactions. In such a case, a useful representation of the dependency knowledge could be in the form of a matrix, where the rows are the different transactions and the columns are the monitored resources. In a simple representation the fact that transaction  $t$  depends on the services of resource  $s$  can be represented by a 0 or 1. The starting point of this is such a 0/1 dependency matrix that is computed using algorithms described in [3][4][10]. A more complete approach would look at a non-binary set of values to encode the *degree* (or *strength*) of dependency. This paper looks at how one can use the dependency matrix to address two important questions that arise in the management of distributed transaction processing systems:

- 1) When an alert or fault indication is received by the management system that a particular transaction type is experiencing degraded performance, e.g. unacceptably long response time, how can the root cause of the problem be rapidly determined?
- 2) Provided that we have found the root cause(s), what steps can be taken to correct the problem?

The paper is organized as follows: In section 2 we give an overview of related work and describe in short a way of computing the dependency matrix. In section 3, we provide a conceptual architecture of the system that we are in the process of building, using the algorithms reported in this paper. In section 4, we provide a formal definition of the problem. In sections 5 and 6 we describe our algorithms for the root cause analysis. In section 7 we conclude the paper by listing a number of problems for future research.

## 2 Related Work

The dependency matrix of a large distributed system can be obtained in a number of ways by using direct or indirect methods [3]. Direct methods rely on a human or a static analysis program to analyze system configuration, installation data, and application code to compute dependencies. Indirect methods operate at runtime and may be intrusive [1][2][6], semi-intrusive [4], or non-intrusive [10] with respect to the operational system in the manner they extract dependencies. The non-intrusive method proposed by Gupta *et al.* [10] uses the activity periods to infer the dependency relationship.  $A$  depends on  $B$  with strength  $p$ , if the probability that the activity period  $[b_1, b_2]$  of a resource  $B$  is contained in the activity period  $[a_1, a_2]$  of a resource  $A$ , i.e.,  $a_1 \leq b_1$  and  $b_2 \leq a_2$ . There are also methods using neural networks [7] or belief networks [18] to automatically generate dynamic dependencies.

The self-healing problem, especially the problem of locating the root cause error as quickly as possible, has attracted a lot of interest in the recent years due to difficulty

in managing very large distributed systems. There are three major approaches: rule-based systems, codebook systems [13], and artificial intelligence systems based on Bayesian networks or neural networks [18][21]. Our solution falls in the codebook approach which was firstly proposed by Kliger *et. al.* [13]. They proposed the construction of a "codebook" with distinguishable ability so that any single failure in the system can be determined by matching the results of the transactions with the entries in the codebook. While the running time of the codebook approach primarily depends on the size of the codebook, Brodie *et. al.* proved that finding the codebook with minimum size is NP-hard [5]. Several heuristics for finding a codebook have been proposed [5][10]. Notice that this is an "offline" version compared with the online problem we study here. Specifically, in the above cited references, when there is an error in the system, *all* the transactions of the codebook are run and the results are compared with the columns in the dependency matrix to determine the root cause. In the present work, based on the current status of the system, we select a transaction to run, the result of the transaction is collected and the system status is update accordingly. We aim to find the root cause error as quickly as possible, i.e., we want to minimize the number of transactions that would need to be run to get our result. A similar problem was also studied by Rish *et. al.* [17], who provide a general framework using information theory.

### 3 System Architecture

The initial architecture of a self-healing system, using the results of our dependency analysis, is illustrated in Figure 1. The components are briefly described below:

- The Distributed System box denotes a typical, multi-tier e-Business system consisting of a user or client layer, web access layer, web application services layer and a backend database layer. The system supports a predefined, fixed set of user transactions types. In our experimental setup, we simulate such a system using the TPC-W benchmark [22], which is a standard setup for building an experimental electronic store-front.
- The Monitoring System in this picture includes the various monitoring agents that are typically deployed in a distributed environment to support the collection and dissemination of performance and availability data to management applications. In our setup, we use two important monitoring agents: 1) an agent that monitors the response time of the transactions, from a user perspective, 2) an agent that monitors the various components within an application server environment, such as servlets, EJB's, etc.[20]. Initially, the data collected by the monitoring agents are fed into the Dependency Analysis engine for it to calculate the dependency matrix. On a continual basis, when the monitoring system detects an unacceptable response time for any of the transactions, it invokes the Self-Healing Engine, so that the latter can orchestrate a set of steps for problem resolution.
- The Dependency Analysis box incorporates our dependency extraction algorithm [10]. It is started when the distributed system becomes operational and is allowed to run for a length of time dependent on the traffic load. It is assumed that within this time period, a large majority of the various different types of user transactions have had the opportunity to execute and, hence, provide the

necessary data to the Dependency Analysis box to compute the dependency matrix, an example of which is provided in the next section.

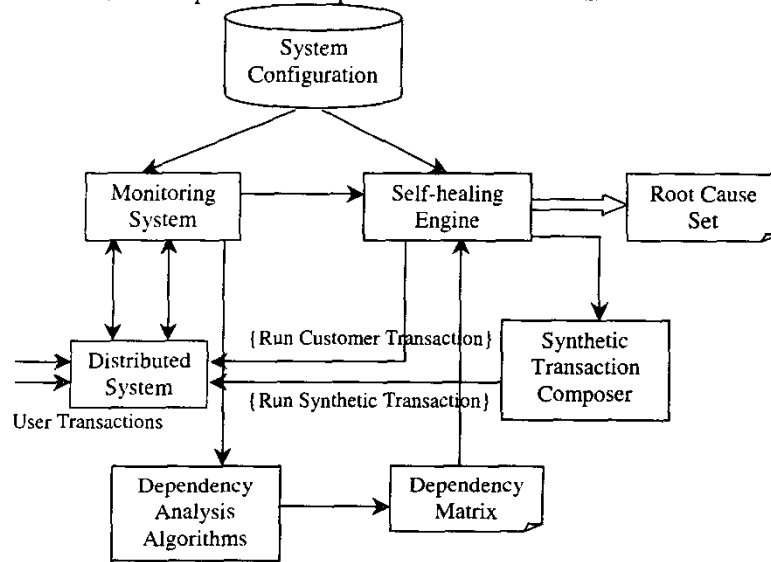


Figure 1: Logical System Architecture

- The Self Healing Engine is the focal point of this paper. It consists of two parts: a **problem determination component** and a **problem resolution component**. In this paper we report on algorithms for the problem determination part. The problem resolution part is an area for future research. On being invoked by the Monitoring System, as a result of a transaction that is experiencing degraded response time, the Self Healing Engine runs one or more algorithms to quickly narrow down the root cause, i.e., the offending resource(s) that is (are) contributing to the degraded response time. The algorithms used in the implementation of the Self Healing Engine, operate in one or both of two ways. It can observe the operations of other transactions in the system and based on their performance narrow down the root cause. Additionally, it may invoke the synthetic transaction composer to construct artificial transactions that can be executed to further eliminate potential candidates for the root cause.
- The Synthetic Transaction Composer consists of a set of pre-canned transactions, one or more of which is selected to run, based on input from the Self-Healing Engine.

In this paper we are going to focus mostly on algorithms that the self-healing engine can use and provide some guidelines for designing the synthetic transaction composer. We assume that the dependency analysis part has been executed and therefore the dependency matrix, is available [1][2][3][4][6][7][10][12][18].

#### 4 Problem statement

Given a large system with a set of resources  $S = \{s_1, s_2, \dots, s_n\}$  and a set of customer transactions  $T = \{t_1, t_2, \dots, t_m\}$ , the dependency of each transaction  $t_i$  on the resource  $s_j$  is represented by a number  $v_{ij}$ . The dependency matrix  $D$  is an  $m \times n$  matrix with the  $(i, j)$  entry being  $v_{ij}$ . We also call the dimension  $n$  vector  $v_i = (v_{i1}, v_{i2}, \dots, v_{in})$  the dependency vector of transaction  $t_i$ . One example of the dependency matrix is as follows:

$$\begin{array}{cccccccc}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 \\
 t_1 & 1 & 0 & 0 & 1 & 1 & 1 & 1 & 0 \\
 t_2 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 1 \\
 t_3 & 0 & 0 & 1 & 0 & 1 & 0 & 1 & 0 \\
 t_4 & 1 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\
 t_5 & 1 & 1 & 0 & 1 & 0 & 1 & 0 & 1
 \end{array} \tag{1}$$

When the execution of a transaction  $t_i$  succeeds, *all* the resources that  $t_i$  depends on are assumed to be working fine. If  $t_i$  fails, then *at least one* of the resources that  $t_i$  depends is assumed to have failed. One way to find the faulty resource(s) from  $S$  which are responsible for causing the transaction to fail is to choose one of the other transactions to run, and based on the result (failure or success), choose the next transaction, etc., until the root cause is determined. Our objective is to minimize the total number of transactions we will need to run to achieve this goal. Before we go into the algorithms, we first look at several issues in the failure detection problem.

1. **0/1 matrix or non-0/1 matrix:** the dependency matrix  $D$  can be a 0/1 matrix, where  $v_{ij} = 1$  means that if  $s_j$  fails then transaction  $t_i$  fails for sure,  $v_{ij} = 0$  means that  $t_i$  doesn't depend on the state of resource  $s_j$ . More generally,  $D$  could also be a non-0/1 matrix, for example, each entry  $v_{ij}$  could denote the conditional probability that transaction  $t_i$  fails given that resource  $s_j$  fails. The entries can also represent quantities other than conditional probability. In this paper we focus on the 0/1 dependency matrix case. Extension of our work to non-0/1 matrix will be reported in the future.
2. **Single failure or multiple failures:** the number of resources that fail at the same time can be one or more. In practice, there is little loss of generality if we assume that only one resource fails at a time. If there could be more than one resource failure, the resources could be either independent of each other, meaning that if  $s_1$  and  $s_2$  by itself works fine, then the combination of  $s_1$  and  $s_2$  is also working fine. A more realistic assumption is that the combination of two resources may fail even if each of the resources is functioning properly. In this paper, we concentrate on single failures, since in practical transaction processing systems a single failure will typically invoke the problem determination system.
3. **Fixed set of transactions or synthetic transactions:** the set of transactions  $T$  is fixed ahead of time, meaning that the users have no freedom to compose new transactions. If the users are given this extra power the problem becomes much easier as we will show later. However, synthetic transactions require programming and impose additional costs.
4. **Zero knowledge or prior knowledge:** if we have no prior knowledge about the system state, we assume that every resource in the system has equal probability

of failure. In many practical cases, we can get prior knowledge about the system state by studying log files and management metric variables, i.e., we have a probability distribution  $P=\{p_1, p_2, \dots, p_n\}$  on the resources in the system, with  $p_i$  representing the failure likelihood of resource  $s_i$ .

The two most common scenarios that we focus on in this paper are as follows. They naturally map to the Self Healing Engine and the Synthetic Transaction Composer in Figure 1.

1. **Single Failure Detection Problem w/ fixed transactions (SFD1):** there is a single failure in the system. The set of transactions one can use to test the system is fixed, as is typical in a transaction system that implements a standard electronic store front. We have zero-knowledge or partial knowledge about the system status. The dependency matrix is 0/1-matrix. The goal is to minimize the number of transactions that need to be run to identify the failed resource.
2. **Single Failure Detection Problem w/ synthetic transactions (SFD2):** we can synthesize new transactions. This could be a transaction processing system, e.g., a financial clearing house system, where application programmers have more freedom in creating test transactions. The other assumptions are the same as above.

#### 4.1 Our results

Our results are listed as follows,

1. SFD1 is NP-hard, even in the offline version where we assume the all the transactions are known *a priori*.
2. Any online algorithm to solve SFD1, in the worst case, runs a factor of  $\Omega(n)$  transactions more compared with the optimal (smallest) set of transactions in the static setting.
3. The worst case happens when the dependency matrix has specific characteristics. In practice, the dependency matrix is much better than the one corresponding to the worst case scenario. Therefore, we propose heuristic algorithms for both SFD1 and SFD2.

### 5 Single Failure Detection Problem w/ fixed transactions (SFD1)

We assume that there is only one failed resource in the system and the resources are independent of each other. We first look at the offline version of the SFD1 problem. We are given a set of transactions  $T$  and we need to conclude from their performance (for example, response time) which resource in the system is faulty. First we should observe that there aren't always solutions to this problem. For example, if the dependency matrix  $D$  is 1 everywhere, we have no information to pin down the root cause of the error. To guarantee a unique solution to the failure detection problem, no two column vectors of the matrix  $D$  can be the same. This condition is sufficient and necessary as we will show later. (Note that the case of  $D$  being 1 everywhere is a special case of the non-identical columns case.)

Since one transaction failed before the Self Healing Engine is invoked, the resources in the system that the failed transaction depends on must contain an error. Denote the set of suspicious resources as  $S$ . The problem is to pin down the root cause in  $S$ , in other words, identify the resource that is the cause of the problem. The algorithm works as follows. We take the set of transactions  $T$  and run all of them. Only a

subset of transactions fails. Take transactions from  $T$  one by one, if  $t_i$  succeeds, we update  $S$  as  $S - \{s_j \mid v_{ij}=1\}$ , or, alternatively, set  $S$  to  $S \cap \{s_j \mid v_{ij}=0\}$ ; if  $t_i$  fails, we update  $S$  as  $S \cap \{s_j \mid v_{ij}=1\}$ . When  $S$  contains only one resource, we have pinpointed the faulty one. If  $S$  contains more than one resource after we run out of transactions, then there is not enough information to narrow down the root cause. One should either consider more transactions or use synthetic transactions. The running time of the algorithm is linear to the number of resources in the system. We now prove the correctness of the algorithm, i.e., by the time we try all the transactions from  $T$ , there is only one resource left in  $S$  if no two column vectors of the matrix  $D$  are the same. Assume otherwise, i.e., we have two resources  $s_j, s_k$  in the set  $S$ . Then by the algorithm we know that for any transaction  $t_i$  that succeeds,  $v_{ij}$  and  $v_{jk}$  are both 0; for any transaction  $t_i$  that fails,  $v_{ij}$  and  $v_{jk}$  are both 1. Then in the dependency matrix  $D$  we should have the columns corresponding to the resource  $s_j$  and  $s_k$  identical. This contradicts our starting assumption.

### 5.1 SFD1 is NP-hard

In this section, we show that the failure detection problem, i.e., minimizing the set of transactions to run in order to pin down the root cause, is NP-hard. In fact, we prove a stronger result than our original setup: even in the offline version where we assume the results of the transactions are known, selecting the minimum number of transactions to pin down the error is NP-hard.

Assume we have run all the transactions and know which of them succeed/fail. We want to pick up a subset  $U$  of transactions which uniquely determines the failed resource. The goal is to minimize the number of transactions in  $U$ . For each transaction  $t_i$ , in the set of all transactions, we define  $S_i$  to be  $\{s_j \mid v_{ij}=0\}$  if  $t_i$  succeeds, and  $\{s_j \mid v_{ij}=1\}$  if  $t_i$  fails. Then the failed resource  $s_j$  is the common intersection of all the  $S_i$ 's. Now the problem becomes: for a set of  $n$  resources  $S$  and  $m$  sets  $\Pi = \{S_1, S_2, \dots, S_m\}$ , with  $S_i \subseteq S$ , find a minimum subset  $\Pi' \subseteq \Pi$  so that  $\bigcap_{S_i \in \Pi'} S_i = \{s_j\}$ . Then

$U = \{t_i \mid S_i \in \Pi'\}$ . The problem can be further reduced. Assume resource  $s_j$ ,  $1 \leq j \leq n$ , is the failed resource. We take out  $s_j$  from each set  $S_i$ , now we want to find a minimum subset  $\Pi' \subseteq \Pi$  so that  $\bigcap_{S_i \in \Pi'} S_i = \emptyset$ .

**Theorem: Finding the minimum number of transactions which uniquely determine the failed resource is NP-hard.**

*Proof:* This is shown by reducing this problem to the Set Cover problem which is known to be NP-hard [9]. The Set Cover problem assumes a set of elements  $S$  and  $m$  sets  $\Pi = \{S_1, S_2, \dots, S_m\}$ , with  $S_i \subseteq S$ , find the minimum subsets  $\Pi' \subseteq \Pi$  so that  $\bigcup_{S_i \in \Pi'} S_i = S$ . For each subset  $S_i$ , we define its inverse  $\bar{S}_i = S - S_i$ , then we want to

find a minimum subsets  $\bar{\Pi}' \subseteq \bar{\Pi} = \{\bar{S}_1, \dots, \bar{S}_m\}$  so that  $\bigcap_{\bar{S}_i \in \bar{\Pi}'} \bar{S}_i = \emptyset$ . This is exactly the

problem of finding the minimum number of transactions that uniquely decide the failed resource.

Since our problem is equivalent to the Set Cover problem, we can adapt the approximation algorithm for the Set Cover problem. The greedy algorithm works as

follows: we initialize  $\Pi'$  as an empty set. Define  $V = \bigcup_{S_i \in \Pi'} \overline{S_i}$ . If  $V=S$ , then we are

done with a set of transactions  $\Pi'$ . If  $V \neq S$ , select the set  $S_i$  so that  $|V \cup \overline{S_i}| - |V|$  is the maximum among all the remaining  $S_i$ . The greedy algorithm has an approximation factor of  $1 + \ln n$  [11]. And it's all we can hope for because getting a better approximation factor is also NP-hard [16]. In plain English, the algorithm is,

1. Choose that transaction from the set of all transactions such that the set of possibly failed resource is the smallest.
2. Choose the next transaction in a way such that the above set can be reduced by the maximum number of resources.
3. Repeat step 2 until the set of possibly failed resources cannot be reduced further.

## 5.2 On-line SFD1 Problem

Assume the status of the transactions is not known, so we want to choose the transactions and run them one by one until we can determine the failed resource. The goal is to minimize the number of transactions we need to run, so that the root cause of the failure can be determined rapidly. We compare the performance with the optimal off-line solution in terms of competitive ratio, which is defined as the number of transactions obtained by our solution compared with the best offline solution. The first observation is discouraging (in terms of competitive ratio). Compared with the optimum offline solution, *any* online algorithm (deterministic or randomized) can have a competitive ratio  $\Omega(n)$  in the worst case, where  $n$  is the number of resources. Assume we have an  $m \times n$  dependency matrix as following, where  $m=n+1$  in this special case.

$$D = \begin{matrix} & t_1 & t_2 & t_3 & \vdots & t_{n+1} \\ \begin{matrix} t_1 \\ t_2 \\ t_3 \\ \vdots \\ t_{n+1} \end{matrix} & \begin{pmatrix} 1 & 1 & 1 & \cdots & 1 \\ 1 & 0 & 0 & \cdots & 0 \\ 0 & 1 & 0 & \cdots & 0 \\ \vdots & \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & 0 & \cdots & 1 \end{pmatrix} \end{matrix} \quad (2)$$

Transaction  $t_1$  depends on all the resources and the other  $n$  transactions depend on one resource each. Assume that transaction  $t_1$  has failed; we want to then run some other transactions to help us decide which resource has failed. Therefore in the worst case one has to run  $(n)$  transactions to determine the root cause: all the first  $n-1$  transactions return "successful" answers and the last transaction fails. Then any online algorithm has to run  $n-1$  transactions before it can discover the failed resource. However, the offline algorithm has all the results of the transactions and can choose the single transaction which uniquely determines the failed resource. So the competitive ratio is  $\Omega(n)$ . Notice that here randomization doesn't help either. The above analysis is based on the worst case scenario. But even for average-case scenario, for example, if the resources fail with equal probability, then the competitive ratio of any online algorithm is still  $\Omega(n)$ : the average number of transactions we need to run is  $(1+2+\dots+n-1+n)/n = (n+1)/2$ .



### 5.3 Greedy on-line algorithm for SFD1

Despite the pessimistic results, the dependency matrices in real e-business systems are not like the worst case examples most of time. There is, typically, a lot of overlap between the resources that the transactions depend on. So the worst case as described in the previous section may happen very rarely. We, therefore, propose the following heuristic. We first make the zero-knowledge assumption, i.e. each resource is equally likely to have failed. Assume  $S$  contains the possible failed resources, where each resource in  $S$  has probability  $1/|S|$  to be the failed one. Suppose a transaction  $t_i$  depends on  $x$  resources out of the  $k$  resources in  $S$ . Then the probability that  $t_i$  fails is  $x/k$ , and in that case we narrow down the set of possibly wrong resources to a set of  $x$  resources that  $t_i$  depends on. Similarly, the probability that  $t_i$  succeeds is  $(k-x)/k$ , and then only  $k-x$  can possibly go wrong. Therefore, the expected number of resources left after we run  $t_i$  is

$$x \cdot x/k + (k-x) \cdot (k-x)/k = (2x^2 - 2kx + k^2)/k, \quad (3)$$

which has a minimum value of  $k/2$  when  $x=k/2$ . Therefore, when we choose the next transaction, we always choose one which depends on as near to half of the resources from  $S$  as possible. The intuition is that irrespective of whether the transaction fails or not, we are going to rule out half of the possibilities. In other words, we are trying to get as much information as we can from the result of the transactions. Another observation on the performance of this algorithm is that if we can always find a transaction which depends on a fraction of the resources in  $S$  at each step, we can always eliminate a fraction of the resources by running each transaction. So the final running time is going to be  $O(\log n)$ .

However, in the real world we usually have or can acquire, through the monitoring system, additional information about the state of the resources. This information will enable us to associate a probability of failure with each of the suspected resources. Assume we have  $p_j$  associated with each resource  $s_j \in S$  which represents the probability of  $s_j$  being the failed one,  $\sum_{s_j \in S} p_j = 1$ . Assume a transaction  $t_i$  depends on

$x$  resources in  $S$  with the summation of their probabilities as  $d_i$ , by the intuition shown in the algorithm with zero-knowledge, the criterion of choosing the next transaction is to choose the one with  $d_i$  as close as  $1/2$  as possible. After we are done with one transaction, we then rule out those in  $S$  that cannot be wrong and re-normalize the probability  $p_j$  for those that are left. This process is continued until either  $S$  contains only one resource, or we've run out of the transactions. In the later case, where there is more than one resource left in  $S$ , we don't have enough information from the dependency matrix to make further distinction. By some derivation we can show that the intuition is also explained by the entropy method in information theory [14]. See [8] for details. To summarize, the algorithm to SFD1 is,

1. The set  $S$  of possible failed resources is initialized to be the set of resources that the first failed transaction depends on.  $P$  is the probability distribution on  $S$ ,  $p_j$  representing the failure likelihood of resource  $s_j \in S$ .

2. Choose the transaction  $t_i$  so that the summation of the probabilities of the resources that  $t_i$  depends on is the closest to  $1/2$  among all the remaining untested transactions.
3. Run  $t_i$ . If it succeeds, then the resources that  $t_i$  depends on are all working fine. We then change their probabilities of failure to be zero. Otherwise, if  $t_i$  fails, the resources that  $t_i$  depends on must contain the failed resource. We change the probabilities of the resources that  $t_i$  doesn't depend on to zero.
4. Renormalize the probabilities  $P$  so that  $\sum p_i = 1$  for the remaining resources.
5. Repeat step 2 until  $S$  has only one resource or we've run out all customer transactions.  $S$  is the minimal faulty resource set.

Given: a set of possible faulty resources  $S = \{s_1, s_2, \dots, s_n\}$  and a probability distribution  $P = \{p_1, p_2, \dots, p_n\}$ , with  $p_j$  representing the failure likelihood of resource  $s_j \in S$ . A set of transactions  $T = \{t_1, t_2, \dots, t_m\}$  is used to test the resources.

```

while |S| > 1 and |T| > 1 do
  d =  $\sum_{v_j=1} p_j$ , k=1;
  for i=2 to |T|
    d_i =  $\sum_{v_j=1} p_j$ ;
    if |d_i - 1/2| < |d - 1/2| then d = d_i, k = i;
  Run t_k; T = T - {t_k};
  if t_k succeeds
    then for j=1 to n
      if v_kj = 1 then p_j = 0; S = S - {s_j};
      else p_j = p_j / (1 - d);
    else for j=1 to n
      if v_kj = 0 then p_j = 0; S = S - {s_j};
      else p_j = p_j / d;

```

Figure 2: Greedy online algorithm for SFD1.

## 6 Single Failure Detection Problem w/ synthetic transactions (SFD2)

### 6.1 Online SFD2

In the case that the fixed set of customer transactions cannot pin point the root cause, we need to compose synthetic transactions. Construction of synthetic transactions for testing and fault diagnosis is a difficult task, since in a real customer environment, they need to be constructed with care, such that they do not interfere adversely with the operation of the real system. Also, construction of synthetic transactions that involve the participation of any arbitrary subset of the total set of resources in the distributed environment may be impossible to do, given the constraints of the physical system. In our analysis below, we ignore such constraints and assume that our Synthetic Transaction Composer component is able to construct any transaction

to exercise an arbitrary set of resources. In a subsequent report we will address the challenge of incorporating the constraints mentioned above.

The algorithms and analysis for the fixed transactions case work here, except that when we choose the next transaction to run, we choose from among the set of transactions that depend on all possible subset of resources. The zero-knowledge algorithm is easy – we select the next transaction which depends on  $\lfloor n/2 \rfloor$  resources, where  $n$  is the number of suspect resources. The algorithm with prior probability distribution  $P$ , however, is not. The problem of finding the transaction  $t_i$  with  $d_i = \sum_{v_j=1} p_j$  to be the closest to  $1/2$  among all the possible transactions, is

equivalent to the Partition Problem (also called Subset-Sum Problem), which is one of the first six problems known to be NP-Complete [9].

**Theorem: Finding the best transaction with respect to the current failure probability distribution  $P$  is NP-hard.**

*Proof:* Omitted, see [8].

The Partition Problem, admits a pseudo-polynomial algorithm [9]. That is, if the probability  $p_j$  has finite precision, the problem can be solved by dynamic programming with running time  $O(n)$  [15]. The hidden constant factor in the running time, however, is very large (inverse of the precision). Instead, we use a simple greedy algorithm that works in  $O(n \log n)$  time.

1. Sort the probabilities  $p_j$  in decreasing order, so that  $p_j \geq p_{j+1}$ , for all  $1 \leq i \leq n$ .
2. A set  $U$  is initialized as an empty set.
3. Inspect the probabilities one by one, if  $p_i + \sum_{p_j \in U} p_j \leq 1/2$ , add  $p_j$  to  $U$ ;  
otherwise, discard  $p_j$ .
4. Stop when we inspect all  $p_j$ . Output  $U$ .

Given a probability distribution  $P = \{p_1, p_2, \dots, p_n\}$ , find a subset  $U$  with summation close to  $1/2$ .  
Sort the probabilities  $p_j$  in decreasing order.  
 $U = \{\}; p = 0;$   
**for**  $i = 1$  **to**  $n$  .  
    **if**  $p_j + p \leq 1/2$   
        **then**  $U = U + \{ p_j \}; p = p_j + p;$

**Figure 3:** Online algorithm for SFD2.

The approximation ratio of the greedy algorithm is defined as the

ratio  $\alpha = \frac{u^*}{u} = \frac{\sum_{p_j \in U^*} p_j}{\sum_{p_j \in U} p_j}$ , where  $U^*$  is the optimum set.

**Theorem: The greedy algorithm has an approximation ratio at most 2.**

*Proof:* Omitted, see details in [8].

**Theorem: The number of transactions to narrow down the root cause is  $O(\log n)$ , where  $n$  is the number of suspicious resources initially.**

*Proof:* Omitted, see details in [8].

## 7 Conclusion and Future Work

In this paper we have discussed the failure detection problem in a large distributed transaction processing system. In order to build a viable "self healing" transaction processing system, one has to design algorithms that can rapidly determine the root cause of a failed transaction. After the root cause is determined, depending on its type, a variety of corrective measures can be taken. This combination of problem detection and resolution steps, along with a high degree of automation in each, would lead to a "self healing" system.

The starting point of the work reported here is a dependency matrix that captures, for every transaction, the resources that it depends on. For simplicity, we consider only binary dependencies, i.e., a 0/1 matrix. We point out the possibility of using dependency matrices with additional information, such as probability values. Algorithms that utilize such information are being looked at as extensions of the work reported in this paper. In typical transaction processing systems, usually when a user transaction manifests degraded response time, there is generally one root cause resource that lies at the heart of the problem. In such cases, if multiple resource failures are noted by the management system, it is usually because the other resources are directly or indirectly dependent on the failed resource. Based on this observation, in this paper we have focused on the situation where transactions fail because of single resource failures. This has allowed us to design and analyze our problem determination algorithms in tractable way, without losing generality.

The root cause identification algorithm is based on a probabilistic approach, where the final set of resources identified as the possible root cause candidates are the ones with the highest probabilities of being so. It is possible, but unlikely, that none of the resources in the set is the actual root cause. It is the authors' belief if such a situation arises, the dependency matrix can be modified to reduce the possibility of such an occurrence in the future. We are basing our analysis on simplistic assumptions in order to make progress and understand the effectiveness of our approach. We believe that as we begin to validate these assumptions with experimental results, some changes in the assumptions as well as in the algorithm will have to be made. The use of dependency matrix in reducing the size of the possible root cause set is a novel approach. Additionally, understanding how to construct synthetic transactions using dependency matrix is a novel contribution.

As continuation of the work reported here, we are planning to consider the following problems:

- Find out the minimum faulty resources set if there are multiple failures at the same time in the system.
- Solve the failure detection problem when the dependency matrix is non-0/1.
- Based on the information generated in the problem detection stage, design appropriate problem resolution algorithms.

### ACKNOWLEDGMENT

The authors wish to thank Manoj Agarwal, Sugata Ghosal, Manish Gupta and Anindya Neogi, IBM India Research Lab., for providing valuable feedback on ideas related to the results presented in this paper.

### References

- [1] J. Aman, Ç.K. Eilert, D. Emmes, P. Yocom and D. Dillenberger, "Adaptive Algorithms for managing a distributed data processing workload", *IBM Systems Journal*, vol.36, no.2, 1997.
- [2] "Systems Management: Application Response Measurement", *OpenGroup Technical Standard C807*, UK ISBN 1-85912-211-6, July 1998, <http://www.opengroup.org/products/publications/catalog/c807.htm>.
- [3] S. Bagchi, G. Kar and J.L. Hellerstein, "Dependency Analysis in Distributed Systems using Fault Injection: Application to Problem Determination in an e-commerce Environment" *12th International Workshop on Distributed Systems: Operations & Management*, 2001.
- [4] A. Brown, G. Kar, and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in Distributed Environment", *IM* 2001.
- [5] M. Brodie, I. Rish, S. Ma, A. Beygelzimer and N. Odintsova, "Strategies for Problem Determination Using Probing". *IBM Technical Report*, 2002.
- [6] M.Y. Chen, E. Kiciman, E. Fratkin, A. Fox, and E. Brewer, "Pinpoint: Problem Determination in Large, Dynamic Internet Services", *International Conference on Dependable Systems and Networks (DSN'02)*, June 2002.
- [7] Ensel, Christian, "New Approach for Automated Generation of Service Dependency Models" *Second Latin American Network Operation and Management Symposium, LANOMS*, 2001.
- [8] J. Gao, G. Kar, P. Kermani, M. Agarwal, S. Ghosal and A. Neogi, Approaches to Building Self Healing Systems using Dependency Analysis, *IBM Research Report*, 2003.
- [9] M. R. Garey and D. S. Johnson, *Computers and Intractability: A Guide to the Theory of NP-Completeness*, W. H. Freeman And Company, New York, 1979.
- [10] M. Gupta and A. Neogi and M. K. Agarwal and G. Kar, "Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination", *IEEE/IFIP International Workshop on Distributed Systems Operations and Management (DSOM)*, Heidelberg, Germany, 2003, in press.
- [11] D. S. Johnson, "Approximation Algorithms for Combinatorial Problems", *J. Comput. System Sci.* 9, 256-278, 1974.
- [12] A. Keller and G. Kar, "Classification and Computation of Dependencies for Distributed Management", *5<sup>th</sup> IEEE Symposium on Computers and Communications (ISCC)*, July 2000.
- [13] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie and S. Stolfo, "A Coding Approach to Event Correlation", *Proc. 4th International Symposium on Integrated Network Management (IFIP/IEEE)*, May 1995.
- [14] D. J.C. MacKay, *Information Theory, Inference, and Learning Algorithms*, Cambridge University Press, 2003.
- [15] D. Psinger, An  $O(n)$  Algorithm for the Subset Sum Problem, *DIKU, University of Copenhagen, Denmark, Report 95/6*, 1995.
- [16] R. Raz and S. Safra, "A Sub-constant Error-probability Low-degree Test, and Sub-constant Error-probability PCP Characterization of NP", *Proc. 29th Ann. ACM Symp. on Theory of Comp.*, ACM press, 475-484, 1997.
- [17] I. Rish, M. Brodie, N. Odintsova, S. Ma and G. Grabarnik, "Problem Determination via Active Probing", *manuscript*, 2003.

- [18] J. W. Sheppard and W. R. Simpson, "Improving the Accuracy of Diagnostics Provided by Fault Dictionaries", *Proceedings of the 14<sup>th</sup> IEEE VLSI Test Symposium*, 1999.
- [19] <http://www.tivoli.com>.
- [20] <http://www-3.ibm.com/software/webservers/>
- [21] H. Wietgrefe, K-D Tuchs, K. Jobmann, G. Carls, P. Froelich, W. Nejd and S. Steinfeld. "Using Neural Networks for Alarm Correlation in Cellular Phone Networks", *International Workshop on Applications of Neural Networks to Telecommunications*, 1997.
- [22] TPCW Wisconsin website, <http://www.ece.wisc.edu/~pharm/tpcw.shtml>.