Dependency Analysis in Distributed Systems using Fault Injection: Application to Problem Determination in an e-commerce Environment

Saurabh Bagchi[+], Gautam Kar, Joe Hellerstein
IBM Thomas J. Watson Research Center
Hawthorne, New York
{sbagchi, gkar, hellers}@us.ibm.com

Distributed networked applications that are being deployed in enterprise settings, increasingly rely on a large number of heterogeneous hardware and software components for providing end-to-end services. In such settings, the issue of problem diagnosis becomes vitally important, in order to minimize system outages and improve system availability. This motivates interest in dependency characterization among the different components in distributed application environments. A promising approach for obtaining dynamic dependency information is the Active Dependency Discovery technique in which a dependency graph of e-commerce transactions on hardware and software components in the system is built by individually "perturbing" the system components during a testing phase and collecting measurements corresponding to the external behavior of the system.

In this paper, we propose using fault injection as the perturbation tool for dynamic dependency discovery and problem determination. We describe a method for characterizing dependencies of transactions on the system resources in a typical e-commerce environment, and show how it can aid in problem diagnosis. The method is applied to an application server middleware platform, running end-user activity composed of TPC-W transactions. Representative fault models for such an environment, that can be used to construct the fault injection campaign, are also presented.

Keywords*:* Problem determination, Dynamic dependency characterization, Fault injection, E-commerce environment, WebSphere Application Server.

1. Introduction

Increasingly, distributed networked applications being deployed in enterprise settings rely on a large number of heterogeneous hardware and software components for providing end-to-end services. Examples of such settings are expected to abound in the next generation of e-business systems, which will be constructed out of service building blocks, possibly provided by different service providers and interacting among themselves to provide tailored services to the end-user. In such settings, the issue of problem diagnosis becomes vitally important in order to minimize system outages and improve system availability. Problem determination approaches in current distributed environments are often ad hoc in nature. The lack of good diagnosis tools leads to enormous loss of time and consequent loss of revenue for the consumers of such distributed application environments. For example, a study commissioned by Stratus Computers found that in the trading and investment banking sector, an hour of system outage would cause a revenue loss of 6 million dollars on an average [10].

A promising approach for problem determination in large systems is dependency analysis. In brief, the question that dependency analysis tries to answer is this: Is the service X dependent on another

[+] Contact Author: Saurabh Bagchi, IBM Thomas J. Watson Research Center, 30 Saw Mill River Road, Hawthorne, NY 10532, USA. Phone: +1-914-784-7816. Fax: +1-914-784-7455.

O. Festor and A. Pras (Eds.) : 12th International Worshop on Distributed Systems: Operations and Management DSOM'2001 Nancy France, October 15-17, 2001.

service Y or resource Z? If such a dependency exists, what is the strength[1] of the dependency? Using this information, when a problem is observed at a particular service point, the root cause may be tracked down to a resource on which this service is dependent. The dependency analysis problem becomes very challenging in situations where the resources in the system are dynamic in nature. In such cases, resources can appear and disappear during system lifetime because of failures, or deployment of new sub-systems, and the dependency relations can change as a result of change of resource availability or new service level agreements being negotiated.

Another valuable approach for problem characterization in systems is fault injection. Fault injection enables accelerated testing of the system under stressful conditions and can help uncover design and implementation defects in the system. A large volume of work in the area has provided us with an extensive set of fault models targeted towards software and hardware components, starting from the simple bit flip model to complex timing faults in software systems.

In this paper, we propose a method for using fault injection to uncover resource dependencies in a dynamic distributed e-commerce environment. The paper also presents the concept of fault dictionaries for a representative e-commerce environment. A fault dictionary provides a mapping from the high-level faults commonly observed in such an environment to the low-level faults that can be directly injected into the resources comprising the system. We present the design of a fault injection campaign manager that uses incomplete knowledge of the system dependencies and fault dictionaries to produce a fault injection campaign. During the execution of the campaign, external probes collect measurements of the relevant metrics, like response time, and the measurements reveal actual resource dependencies in the system. The proposed method is applied to a typical three-tier web-based e-commerce environment with IBM's WebSphere being used as the application server in the target system. The system is used to run a typical end-user e-commerce activity initiated through a web browser and consisting of several transactions taken from the TPC-W benchmark [13].

The rest of the paper is organized as follows. Section 2 presents the background for the current work, mentioning previous work in the area of dependency analysis and fault injection. Section 3 presents the algorithm. Section 4 presents the prototype environment on which the method is to be applied and a concrete example of its application. Section 5 presents a discussion of the applicability of the technique and concludes the paper.

2. Background

One of the most complex tasks performed in the management of enterprise distributed systems is problem determination and resolution - detecting system problems, isolating their root causes and identifying proper repair procedures. A promising approach lies in the design of problem detection algorithms that use dependencies between software and hardware components that constitute a distributed system. Much work is present in the literature describing the use of dependency models for the important root-cause analysis stage of problem determination, that is, for the process of determining which system component is ultimately responsible for the symptoms of a given problem. However, there has been little work on the important problem of automatically obtaining accurate, detailed, and up-to-date dependency models from a complex distributed system.

A distributed environment can be logically modeled as layers of resources – services, applications and other software and hardware components – that cooperate to deliver an end-to-end service. Services or components in one layer depend on functions provided by components in a lower, supporting layer. Failures occurring in one layer affect the functioning of dependent components in another layer. The premise is to model a system as a directed, acyclic graph in which nodes represent system components (services, applications, OS software, hardware, networks) and weighted directed edges represent dependencies between nodes. A dependency edge drawn between two nodes indicates that a failure or problem with the node at the head of the edge can affect the node at the tail of the edge. The weight of the edge represents the impact of the head node's failure on the tail node. The dependency graph for a heavily simplified e-commerce environment is depicted in Figure 1.

[1] A metric that measures the effect of the dependency on the performance of a dependent service

Strength of the dependencies is denoted by Strong (S), Medium (M) and Weak (W), concepts that are explained in detail in Section 3.2.

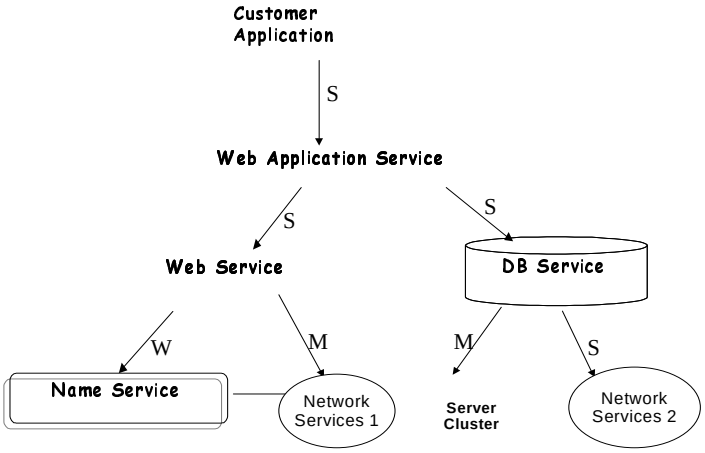

Figure 1. A sample dependency graph. A service entity at the tail of an edge depends on the entity at the head of the edge.

Consider that we are able to capture the dependency information in the above form. Then, the task of identifying the root cause of a problem can be approached by navigating through the dependency graph. However, in practice, the difficulty lies in the fact that service dependencies are not made explicit in today's systems, thus necessitating the task of discovering the dependencies. What is needed is a dynamic model reflecting the changing dependency relationships between services. A well-known approach, called static dependency analysis, uses information present in various configuration files describing a distributed system, to derive dependency information [8]. Experience indicates that such an approach, while it can provide some single node, or intra-domain dependency information, fails to capture the entire dependency picture, particularly when the dependency crosses domain boundaries. The limitation of the technique arises due to the contents of configuration files not being standardized and containing only incomplete and sometimes outdated information on dependencies.

A second approach, called *Active Dependency Discovery* (ADD), addresses this shortcoming by actively perturbing system components while monitoring the system's response [2]. For example, in a database system, the perturbation can take the form of holding on to a lock to a database table and not releasing it for a certain interval of time. If response time to a database operation is an important QoS property in such a system, then the measured metric can be the time to complete a transaction in the presence of perturbations. Statistical regression analysis on collected data on the output metric and the degree of perturbation enables one to fit regression lines indicating the presence and strength of dependencies of the output QoS metric on the components that had been perturbed. An advantage of the technique is its ability to differentiate causal relationships indicating actual resource dependencies from simple correlations in monitoring data since there is knowledge of which component is being perturbed. However, the coverage of the approach is dependent on the ability to insert controlled perturbation to the different system components. While this technique is considerably more successful in discovering dependencies, it is reliant on the use of appropriate faults that can effectively uncover dependencies. This paper addresses the problem of designing a system that can identify the appropriate faults that can be injected into a distributed system, leading to the discovery of dependencies between components of the system. Also, the ADD approach does not address the problem of injecting correlated faults in multiple system components, while real-world data shows that most system failures occur due to the occurrence of multiple correlated faults [4].

Fault injection is considered an important tool for evaluating the dependability of computer systems. Faults are injected into the system to study the dependability bottlenecks, to study the

behavior of the system under fault conditions, and evaluate the effectiveness and performance impact of fault tolerance mechanisms, namely, the error detection and recovery mechanisms. Many fault injection tools have been developed both commercially and in universities. The tools may be hardware-based which use additional hardware to introduce faults into the target system's hardware, or they may be software-based which involve inserting the fault injection module in the application software or the operating system or in the layer between the two. Some examples of hardware fault injectors are Messaline from LAAS-CNRS in Toulouse, France [1] and MARS developed at the Technical University of Vienna [7]. Software fault injectors are more flexible in the types of faults they can inject and the targets they can inject faults to, and are also generally less costly to develop. Software fault injection tools greatly outnumber the hardware tools. Some examples are NFTAPE from the University of Illinois [11], Ferrari developed at the University of Texas at Austin [9] and Xception from the University of Coimbra in Portugal [3]. A good overview of the field of fault injection is to be found in [5].

The fault injectors are characterized by the different fault types, fault locations and fault triggers that they can support. Fault types can be permanent or transient, and can follow different models, such as pin stuck-at faults, message corruptions, etc. The injector may be able to inject faults at different locations, e.g., to the chip pins, the network interface card, the application or the system software, etc. The fault triggers can be specified in terms of time, specific point in the execution of software code, events in a system such as message arrival, etc. A crucial criterion that determines the choice of the fault injector to evaluate a system is an understanding of the kinds of faults likely to be observed in the operational system. Using faults from one single level of abstraction (e.g., low-level faults that affect the pins of the processor chip, or high-level faults such as synchronization faults affecting a multi-threaded application) limits the flexibility of the fault injection technique as a dependability evaluation tool since faults from different levels may be more realistic, or more feasible to inject under different scenarios.

A hierarchical fault modeling approach for system dependability evaluation has been developed by Iyer *et al* [6]. In this technique, the effects of low-level faults (transistor or chip level) are propagated to higher levels (system level) using fault dictionaries. In our current work, we make use of several concepts from this earlier work, namely, Low Level Faults (LLF), Lightweight Fault Injector (LWFI), Fault Dictionary (FD) and Integrated Fault Injection Campaign (IFIC). The NFTAPE automated fault injection environment [12], which is based on these concepts, is chosen as the tool for the current study.

Low-level faults refer to faults at the level of transistor, chip or memory bits which are close to the manifestations of actual physical faults (as opposed to high-level faults which are indirect manifestations of complex faults in application or system software). A fault injector whose only function is to inject faults and which relies on other services for the common tasks like communication, workload generation, fault triggering, logging, etc., is called a *Lightweight Fault Injector* (LWFI). A *fault dictionary* details the impact of faults on some subset of the target system in terms of the resulting change in the subsystem's behavior as seen from outside the subsystem. For example, a current surge in a particular transistor of an adder circuitry can result in a wrong data value being output by the adder chip. In a fault dictionary, theoretically, knowing any low-level fault (such as, current spikes) and its location and temporal information, one can look up the corresponding entry to find what higher-level fault it causes. This can provide a very powerful tool in dependability evaluation of our target dynamic distributed systems. Having constructed the fault dictionary for such a system (or certain components of the system), and knowing the low-level fault environment, one can develop and deploy fault injectors to inject high-level faults. The rationale behind this approach is that it is usually easier to obtain the characteristics of lower level faults, while fault injection can be accelerated if the injection is performed using higher level fault models. An *Integrated Fault Injection Campaign* consists of a listing of the LWFIs, their targets, their trigger conditions, and a specification of the concurrency of their execution.

To our knowledge, this paper presents the first attempt at using fault injection for dependency characterization in distributed e-commerce systems. It provides a taxonomy of high-level faults

typically observed in e-commerce environments and uses fault dictionaries to derive low-level injectable faults from them.

3. Dependency Analysis Method

3.1. Logical Structure of an E-Commerce Environment

Consider the logical structure of a typical three tier web-enabled e-commerce environment, depicted in Figure 2. In the setup, a typical end-user activity is initiated through a web browser client. A sample activity comprises visiting an internet store front, searching for some items, and finally making the purchase of one or more items. The activity is decomposed into several transactions each of which obeys the traditional ACID properties – atomicity, consistency, isolation, and durability. Examples of transactions are registering a user name and password, doing a database query for a particular item in the store, validating a credit card for a buying activity, etc. The transactions are initiated by sending requests to the web server. The web server invokes the servlet engine for processing the request, and the servlet engine spawns servlets, which implement the business logic. The servlets use tables stored in a database at the backend for processing the request. Finally, the response for the transaction is communicated back to the end-user by the web server.

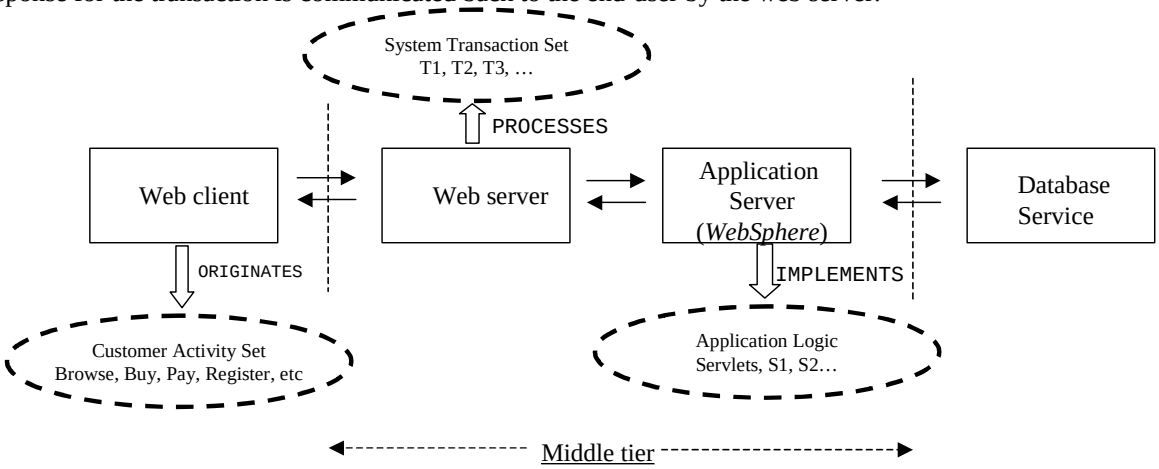

Figure 2. Model e-commerce environment to be used for applying dynamic dependency characterization method

3.2. Dependencies in an E-commerce environment

First we define some concepts and some ground rules for the environment under consideration. An *activity* is defined as a high-level task initiated by the end-user through the web client. The activity will cause a set of transactions to be submitted to the middle tier of the environment. The successful completion of the activity is dependent on each of the constituent transactions completing successfully. The transactions that are processed in an e-commerce environment depend on a set of hardware and software resources, e.g., database tables, network connectivity, a key server for providing keys for a secure transaction, etc. This set of resources for a transaction, Ti is called the *Transaction Dependency Set* (T_iD). The dependency relation of a transaction on a resource can be *absent(A)*, *weak(W)*, *medium(M)*, or *strong(S)*. Our research shows that the strength metric is dependent on the context for which we are planning to use this information.. For example, the metric may have one set of values for fault and availability management and a different set of values for performance management. In the e-commerce example that we have used in our research, we have assigned the values based primarily on performance management dealing with response time related metrics. The relation between transactions and resources can be conveniently represented as a matrix (Table 1) with the cells indicating the strength of the dependency. A similar representation was introduced in [2].

	Transaction T1	Transaction T2
Resource R1	A	M
Resource R2	S	S
Resource R3	W	A

Table 1. Matrix representing dependency relation of transactions on resources.
(A = Absent, W = Weak, M = Medium, S = Strong)

The *difference* between the dependencies of two transactions on a resource is given a measure which follows the relation shown in the lattice in Figure 3. For example, if (S-A) is given a measure of 5 units, (M-A) can be given a measure of anything less than 5. Say, it is given a measure of 3 units. The (M-A) difference does not constrain the assignment of the (S-M) difference since there is no edge between the two nodes. The exact assignment of the differences is governed by the specific details of the system. A measure called *closeness* is defined between every pair of transactions as the sum of the differences over all the resources in the environment.

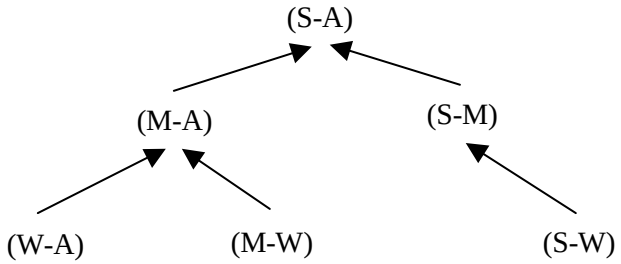

Figure 3. Lattice representing difference between degrees of dependency

3.3. Protocol Details

The high-level steps to be followed in the process of dependency characterization of a distributed system using fault injectors as perturbors is described below. The steps also outline how the dependability characterization can finally lead to problem determination.

1. Suppose the QoS of an activity X is found to violate the service level agreement (SLA). Suppose that it is determinable through an application characterization that activity X uses transactions $T(X) = \{TX_1, TX_2, ..., TX_n\}$. For example, a buy activity will require establishment of a secure session with the client, typically through the Secure Sockets Layer (SSL) session-level protocol. We choose other activities (Y1, Y2, ..., Yn) from the workload set such that $T(X) \cap T(Y1) \cap ... T(Yn)$ is as *close* to T(X) as possible, by the definition of closeness in section 3.2. How such activities Y1, Y2, ..., Yn are chosen is a topic of ongoing research and is outside the scope of this discussion. For the purposes of the rest of the paper, we will assume that each activity consists of a single transaction, thus making the selection of the set simple. The number of such activities chosen, *n*, is a parameter to this process and depends on how much time is available for the process and the confidence to be placed on the process' output.

2. Run activities Y1, Y2, ..., Yn and mark the ones that are also found to have the problem, i.e., violate the SLA. Say, these activities are Z1, Z2, ..., Zk. Now consider the transactions that are common to all the problematic activities, including X. Then, the Suspect Transaction Set (STS) = $T(X) \cap T(Z1) \cap ... \cap T(Zk)$.

3. Consider one transaction T_i from STS. From an incomplete dependency characterization (through ADD, or static dependency knowledge, say), it is known that T_i depends on a certain set of hardware and software resources in the system, H1, H2, ..., Hm, and S1, S2, ..., Sn.. This is denoted by T_iD. If no such prior dependency analysis is available, then we start by including all the hardware and software resources in the system in this set. The set will be successively pruned

in the following steps and therefore, the method will still be able to identify the dependency though it will take longer to reach termination.

4. The set of resources on which transaction T_i depends is put in the Diagnosis Set (DS), which is the set of resources which are potentially malfunctioning. The set NDS, which is the set of resources that have been determined to be functioning correctly is initially empty. Now consider a set of faults that we would like to perturb the resources with. The set will be heterogeneous since there are different kinds of resources. Say the function mapping the resource to the faults in the set is LLF (for low level fault). We denote the function LLF since it is generally assumed that we can determine more accurately the fault characteristics of the low-level faults, i.e., it is easier to say how many bit flips in memory there are likely to be than how many lock release errors in a software module. However, this assumption is not required for the process to work, it is simply used as a basis for the nomenclature. For the purpose of the fault injectors which will be used to insert the perturbation in the system, we have to arrive at the fault model at the appropriate level at which the lightweight fault injector (LWFI) is available. For this purpose, we use the fault dictionary for the system (FD) to scale the postulated fault model *up* (e.g., from the transistor level to the chip level) or *down* (e.g., from the application level to the chip level). So, the LWFI for the hardware resource H1 in the set DS is given by LWFI(H1) = FD(LLF(H1)). There is also the issue of degree of perturbation of the fault injector. By degree, we mean what is the rate of fault injections, e.g., how many memory bit flips per second are introduced, or how many messages (one out of every n) are corrupted on the network link. For the first fault injection campaign, the degree of perturbation is denoted by d_1. Therefore, the fault injector for H1 becomes $LWFI_{d1}(H1)$. An Integrated Fault Injection Campaign (IFIC) that incorporates the fault injectors for all the resources in DS is given by $IFIC1 = LWFI_{d1}(H1) \bowtie LWFI_{d1}(H2) \bowtie ... \bowtie LWFI_{d1}(Hm) \bowtie LWFI_{d1}(S1) \bowtie LWFI_{d1}(S2) \bowtie ... \bowtie LWFI_{d1}(Sn)$. The symbol $\bowtie$ denotes the coordination between the various LWFIs, e.g., the temporal relations between their triggers that can be specified through NFTAPE [12]. $IFIC_1$ is run for a certain length of time and the output QoS metric(s) measured. A series of successive integrated fault injection campaigns are run by varying the degree of perturbation. The series is given by $IFIC_1$, $IFIC_2$, ..., $IFIC_k$. The objective is to prune the DS (Diagnosis Set) for the transaction T_i. Once the DS cardinality decreases by one, terminate the IFIC series, go back to the beginning of this step and repeat. The repetition terminates when the DS cannot be pruned any further, which, for some cases, will be when it becomes empty. If the DS becomes empty, it implies that none of the resources transaction T_i depends on is malfunctioning; the problem lies somewhere else.

5. Go back to step 3 and repeat for every transaction from STS (Suspect Transaction Set). At the end, we will obtain a set of possible malfunctioning resources on which our original activity X was dependant and hence violating its SLA. Note that if step 4 puts a resource in DS of one transaction T_i and NDS of another transaction T_j, we must conclude that the problem was transient and therefore, an offline problem determination cannot work.

The method is summarized through the pseudo-code description in Figure 4.

[1] Application X violates SLA.
Execute similar applications and measure output QoS paramete rs to identify *Suspect Transaction Set* (STS)
STS = {T1, T2, …, Tn}
[2] for each T_h in STS do
[a] Create *Diagnosis Set* (DS) of potentially malfunctioning resources.
Initialize DS with all hardware and software resources T_h depends on.
DS = {H1, H2, …, Hm; S1, S2, …, Sn}
[b] for each H_i in DS do
[i] Pick the fault model (LLF), look up in fault dictionary (FD) to arrive at appropriate level of fault, instantiate *Lightweight Fault Injector* (LWFI) for injection
$LWFI(H_i) = FD(LLF(H_i))$
end do
[c] for each S_j in DS do
[i] Pick the fault model (LLF), look up in fault dictionary (FD) to arrive at appropriate level of fault, instantiate Lightweight Fault Injector (LWFI) for injection
$LWFI(S_j) = FD(LLF(S_j))$
end do
[d] Determine a range for degrees of perturbation DP = {d1, d2, …, dp}
for each d_k in DP do
while DS is being pruned
[i] Form Fault Injection Campaign (FIC) = $(\bowtie LWFId_k(H_i)) \bowtie (\bowtie LWFId_k(S_j))$
[ii] Run campaign and collect measures of QoS parameters
[iii] Prune DS based on measurements
end while
end for
end do
[3] $\cup$ DS is the set of malfunctioning resources
return ($\cup$ DS)

Figure 4. Pseudo-code for fault injection based method for dependency discovery

4. Application of Technique

The above fault injection technique is applied to a realistic e-commerce environment as an aid to problem determination. The following sections discuss the application of the approach. The application is based on the algorithm presented above and is arrived at by instantiating all the unbound variables in the algorithm, such as the transactions and the resources.

4.1. Environment

Consider a typical end-user e-commerce activity initiated through a web browser client. The activity comprises visiting an internet store front, searching for some items, and finally making the purchase of one or more items. The activity is decomposed into several transactions. For our experiment we used the working transaction set provided by the TPC-W benchmark, proposed by the Transaction Processing Performance Council [13]. This set comprises typical transactions that one would observe in a storefront implementation such as an online bookstore. A simplifying assumption from the algorithm presented in Section 3 is that in this application scenario each activity consists of exactly one transaction. In this example, TPC-W uses the following tables in the backend database: *Item, Country, Author, Customer, Orders, Order line, Credit card transaction, Address*.

Our study is based on the setup shown in Figure 5, where we have depicted a typical, transactional e-commerce environment consisting of five key function groups:

- *Core HTTP engine*: receives and dispatches a user transaction to the appropriate servlet that handles the business logic.
- *Servlet engine*: collection of servlets and their execution environment that together support the business logic.

- *Enterprise Java Beans Server engine*: provides session support for the transactions and database connectivity.
- *Database Engine*: houses the back end data base and associated execution environment.
- *Key Server Engine*: provides support for security and authentication.

A typical transaction is dependent on various resources belonging to each of these logical function groups. As an example, we list below a subset of the TPC-W transactions that constitute typical customer activity. Some of the transactions use encryption and are noted as being secure, while others pass information on the network in the clear and are marked as being non-secure.

1. *Customer registration (non-secure)*: returns to the browser a web page, which allows a user to provide the information necessary to register as a known customer or as a new customer and to submit his registration.
2. *Home web page interaction- browse (non-secure)*: returns to the browser a web page which contains links to product lists for new products and for best sellers. This is the initial web interaction requested by all users starting a new user session. It is also a navigation option to most other web pages.
3. *Search request (non-secure):* returns to the browser a web page which allows a user to specify search criteria to find qualifying items.
4. *Search response (non-secure)*: returns to the browser a web page which contains the list of items that match a given search criteria.
5. *Shopping cart web interaction (non-secure):* allows the user to create a new shopping cart, or refresh an already existing cart from an earlier session. It is also used to add new items to the cart, or update existing items.
6. *Buy request (secure):* displays a summary of the items in the shopping cart. The page provides editable fields for entering credit card information and selecting shipping options.
7. *Buy confirm (secure):* transfers the content of the shopping cart into a newly created order for the registered customer and executes a full payment authorization. It then returns to the browser a web page containing the details of the newly created order.
8. *Order inquiry (secure):* returns to the browser a web page which allows a user to provide the information necessary to enter or confirm his identity as a returning customer. This lets the user query about his last order.

4.2. Solution Approach

The first step is to produce a logical function model of the system that allows the enumeration of the resources that support the working of a specific transaction within the e-commerce environment. Figure 5 depicts a subset of the resources that come into play in a typical 3-tier e-commerce setup as shown in Figure 2.

When a fault is reported by a customer in a transaction, the possible root causes are narrowed down to a set of resources on which the transaction depends, using the results obtained by Active Dependency Discovery (ADD). A fault injection system is designed to inject appropriate low level faults into each of these resources to study the behavior of the transaction, in order to eliminate spurious dependencies and to narrow down the root cause to a smaller set of resources. The *partial* matrix of dependencies computed for our TPC-W environment is shown in Table 2 below.

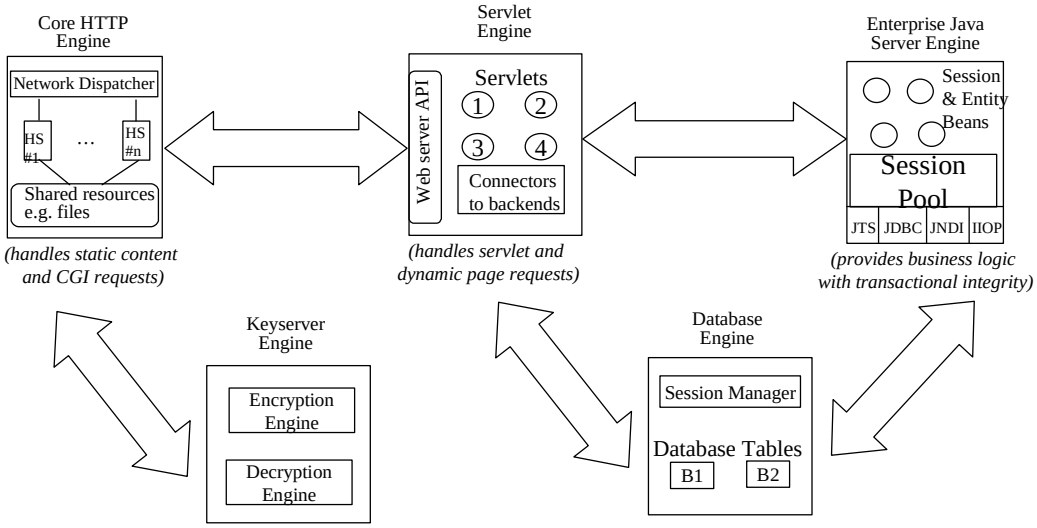

Figure 5. Logical Function Groups in a Typical e-commerce Environment

As an example consider the case where a customer activity – browse an item from an electronic store – causes a problem to occur. This activity consists of one transaction, T2, with an associated dependency set obtained from Table 2. Referring to the method presented in Section 3.3, the Suspect Transaction Set (STS) therefore consists of T2, and the diagnosis set DS = T_2D ={Customer Table, Item Table, Session Pool, Servlet2, Servlet3, DNS}, and NDS = {NULL}. Recall that T_iD is the set of resources that a transaction Ti depends on (see section 3.3, step 3).

	T1	T2	T4	T5	T6	T7
Customer Table	S	S		S	S	S
Item Table		S	S			
Author Table			S			
Order Table				S		S
Address Table	S					S
Order List Table					S	S
Country Table						S
Session Pool		M		W	W	
Servlet1	S					
Servlet2		S				
Servlet3		S		M		
DNS Server	W	W				
Security Server					M	M

Table 2. Partial Dependency Matrix for Experimental TPC-W Environment

We choose a set of transactions "close" to T2 as described in step 1 of the algorithm in Section 3.3. That is, we compute the "closeness" of each transaction in the matrix to T2. For example: Closeness(T2, T5) = (S-S)+(M-W) +(S-M). If this value is less than a predetermined threshold, we include the transaction in the list. Let the final list be T5 and T6. Thus, T_5D = {Customer Table, Order Table, Session Pool, Servlet3} and T_6D = {Customer Table, Order list Table, Session Pool, Security Server}.

TPC-W has drivers to run each of the above transactions individually to study their performance. After doing so we discover that transaction T5 and T6 run with acceptable performance. We

conclude, therefore, that STS = {T2}. As a consequence of this step, the Diagnosis Set is given by, DS = T_2D – [T_5D U T_6D] = {Item Table, Servlet2, DNS} and NDS = {Customer Table, Servlet3, Session Pool}. Thus, at this point, DS contains the most likely root cause of the problem.

Next we generate a fault injection campaign using the system shown in Figure 6.

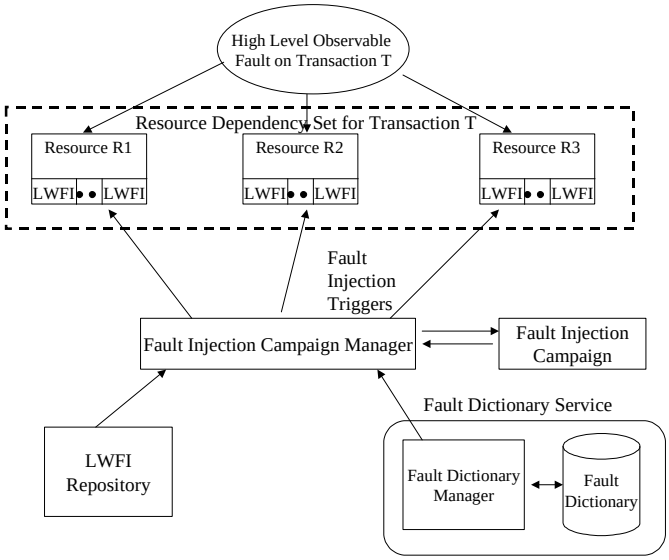

Figure 6. Method for Constructing a Fault Injection Campaign

In our example the three resources, Item Table, Servlet2, and the DNS server correspond to elements in the Resource Dependency set. It is assumed that each of these resources has built-in light weight fault injectors (LWFIs) that can inject faults on receiving triggers from the Fault Injection Campaign Manager. Using the method outlined in Section 3.3, we design a fault injection campaign for each of the elements in the DS. Our approach assumes the presence of a fault dictionary. One of the goals of this research is to come up with a method for constructing an appropriate fault dictionary to aid in the design and implementation of a problem determination system. For our experiment, we start with a hand constructed fault dictionary for the environment under test. A logical structure of such a fault dictionary is shown in Figure 7.

In our example, if the high level observable fault, as reported by the customer was performance related, e.g. a buy transaction takes too long to complete, we find from the table the possible low level faults that can be injected into the resources that remain in the diagnosis set DS. Thus an example fault injection campaign may be expressed as: LWFI (Item Table) = {Memory fault, Data field overrun, Lock contention}, LWFI (Servlet2) = {Resource starvation by reducing the number of threads available in the thread pool, CPU resource starvation by running some other compute-intensive workload on the same processor}, and LWFI (DNS Server) = {Performance fault by submitting high volume of DNS lookup requests on the same server, Corruption of DNS entry}. Each of these faults is injected in turn and the performance of transaction T2 in STS is observed. In the general method, the integrated fault injection campaign can have LWFI's executing concurrently. However, in such cases, it becomes difficult to associate the observed system response to particular faults. Therefore, for simplicity, sequential execution of the LWFI's is considered for the example scenario. The result of this step is to eliminate those dependencies that are not relevant for this transaction, e.g. if servlet3's perturbation does not affect the performance of T2, then we will eliminate it from the DS. Eventually, the DS will be pruned to a set consisting of only those resources that could have been the cause of this problem. Additional diagnostic methods, such as log file analysis, can now be applied to get to the unique root cause of the observed problem.

Observable Fault Table

High Level Fault	**Possible Sources**
Access Denied	Key Server, Database Server
Data not found	Database Server
Performance Degradation	Application Server, Database Server

Resource Fault Table

Resource	**Possible Lower Level Faults**
Database Server	Lock contention, Buffer Size Problem, ……
Application Server	Servelet Error, Application Crash

Fault Injector Table

Resource Fault	**Low Level Fault Injectors**
Application Crash	Process Crash, Memory Corruption

Figure 7. A Representative Three-level Fault Dictionary

The key advantages gained by using our system are twofold:

- Computed a set of suspected resources for problem determination using dependency analysis employing fault injection. This helps to focus the problem determination procedure to a set of possible root causes.
- In typical e-commerce systems, as depicted in Figure 5, the list of dependencies can be quite large, many of them spurious. This step helps speed up the problem determination procedure by rapidly reducing the set of possible root causes.

Additional diagnostics that will need to be performed to get to the root cause can work with only a few resources rather than a large number, resulting in more effective and faster problem determination.

5. Conclusion

This paper discussed the use of well proven fault injection techniques in discovering service and resource dependencies in distributed systems as a step towards efficient problem determination. A method was presented for starting with a large set of potentially malfunctioning resources and successively pruning the set using a set of fault injection campaigns. The method was applied to a real-world web-based e-commerce environment to illustrate how it can aid in root cause determination for an end-user visible problem.

One problem with the approach is that it is invasive in nature and hence needs careful consideration of the conditions under which it can be applied. Can it be applied in an operational system without affecting the normal operation? A notable point here is that the algorithm will be triggered due to a problem indication, such as a SLA violation. Thus, the system is already in a malfunctioning mode. Therefore, if the proposed technique can be executed promptly and the errant resources identified and replaced with backups or hot standbys, the system can be brought back to a

correctly functioning state. However, it is possible that the fault injection campaign is extremely invasive and completely halts the operation of the system. It is a debatable issue whether it is desirable to continue to run a system under malfunctioning conditions when certain QoS guarantees are not being met, or completely halt the system, run diagnosis routines and then bring back a fully operational system as soon as possible. It is most likely that such a decision will have to be made on a case-by-case basis depending on the requirements from the system.

A second point that will determine the applicability of the proposed method is the amount of support available from the system resources which will be made the targets of the lightweight fault injectors. For some types of injectors, little or no support is required from the resource, e.g. to insert a stuck-at-fault at a chip's pin. However, for some other classes of faults, the resource needs to support the injector, e.g., a software fault injector which delays messages in a message-passing based distributed application needs to be compiled in with the application to be able to manipulate the application's message queues. An area of research that could be important in this context is the design of a method that allows an application developer to include LWFI specific to his application during the design and development phases.

As the current research evolves and an implementation is applied to a real-world system, the approach will need to be compared against other approaches to problem determination, such as event correlation. The measures for such a comparison will include the accuracy of diagnosis and the speed of convergence of the technique.

6. References

[1] J. Arlat, Y. Crouzet, J. C. Laprie, "Fault Injection for Dependability Validation of Fault-Tolerant Computer Systems," Proc. 19th International Symp. on Fault-Tolerant Computing (FTCS-19), pp. 348-355, 1989.

[2] A. Brown, G. Kar, A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Application Environment," IEEE/IFIP International Symposium on Integrated Network Management, pp. 377-390, 2001.

[3] J. Carreira, H. Madeira, J.G. Silva, "Xception: Software Fault Injection and Monitoring in Processor Functional Units," Proc. 5th Annual IEEE Int'l Working Conference on Dependable Computing for Critical Applications, pp. 135-149, 1995.

[4] J.B. Dugan, M.R. Lyu, "System-Level Reliability and Sensitivity Analysis for Three Fault-Tolerant System Architectures, Proc. 4th Annual IEEE Int'l Working Conference on Dependable Computing for Critical Applications, pp. 459-477, 1994.

[5] M.C. Hsueh, T.K. Tsai, R.K. Iyer, "Fault Injection Techniques and Tools," IEEE Computer, pp.75-82, April, 1997.

[6] Z. Kalbarczyk, R. K. Iyer, G. L. Ries, J. U. Patel, M. S. Lee, Y. Xiao, "Hierarchical Simulation Approach to Accurate Fault Modeling for System Dependability Evaluation," IEEE Transactions on Software Engineering, Vol. 25, No. 5, pp. 619-632, September/October 1999.

[7] J. Karlsson, J. Arlat, G. Leber, "Application of Three Physical Fault Injection Techniques to the Experimental Assessment of the MARS Architecture," Proc. 5th Annual IEEE Int'l Working Conference on Dependable Computing for Critical Applications, pp. 150-161, 1995.

[8] G. Kar, A. Keller, S. Calo, "Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis," Proc. 7th IEEE/IFIP Network Operations and Management Symposium (NOMS 2000).

[9] G.A. Kanawati, N.A. Kanawati, J.A. Abraham, "FERRARI: A Tool for the Validation of System Dependability Properties," Proc. 22nd International Symp. on Fault-Tolerant Computing (FTCS-22), pp. 336-344, 1992.

[10] D. K. Pradhan, ed., "Fault Tolerant Computer System Design," Prentice-Hall, 1996.

[11] D.T. Stott, M.C. Hsueh, G. Ries, R.K. Iyer, "Dependability Analysis of a High-Speed Network using Software Implemented Fault Injection and Simulated Fault Injection," In IEEE Transactions on Computers, Special Issue on Dependable Computing, pp. 108-119, January 1998.

[12] D.T. Stott, B. Floering, Z. Kalbarczyk, R.K. Iyer, "Dependability Assessment in Distributed Systems with Lightweight Fault Injectors in NFTAPE," Proc. IEEE Int'l Computer Performance and Dependability Symp. (IPDS'2K), pp.91-100, March 2000.

[13] Transaction Processing Performance Council. "TPC Benchmark W, Specification v1.4" San Jose, California, February 7, 2001. Available at http://www.tpc.org/tpcw