An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in a Distributed Environment

A. Brown[*]
Computer Science Division, UC Berkeley
387 Soda Hall #1776
Berkeley, CA 94720-1776, USA
abrown@cs.berkeley.edu

G. Kar, A. Keller
IBM T.J. Watson Research Center
P.O. Box 704
Yorktown Heights, NY 10598, USA
{gkar, alexk}@us.ibm.com

Abstract

We describe a methodology for identifying and characterizing dynamic dependencies between system components in distributed application environments such as e-commerce systems. The methodology relies on active perturbation of the system to identify dependencies and the use of statistical modeling to compute dependency strengths. Unlike more traditional passive techniques, our active approach requires little initial knowledge of the implementation details of the system and has the potential to provide greater coverage and more direct evidence of causality for the dependencies it identifies. We experimentally demonstrate the efficacy of our approach by applying it to a prototypical e-commerce system based on the TPC-W web commerce benchmark, for which the active approach correctly identifies and characterizes 41 of 42 true dependencies out of a potential space of 140 dependencies. Finally, we consider how the dependencies computed by our approach can be used to simplify and guide the task of root-cause analysis, an important part of problem determination.

Keywords

Application Management, Dependency Analysis, Problem Determination

1. Introduction

One of the most significant challenges in managing modern enterprise systems lies in the area of problem determination—detecting system problems, isolating their root causes, and identifying proper repair procedures. Problem determination is crucial for reducing the length of system outages and for quickly mitigating the effects of performance degradations, yet it is becoming an increasingly difficult task as systems grow in complexity. This is especially true as the number of system hardware and software components increases, since more components result in more places that the system manager must examine in order to identify the cause of an end-user-reported problem,

[*] Work done while author was an intern at IBM T.J. Watson Research Center.

and also more paths by which the effects of problems can propagate between components, masking the original root cause.

A promising approach to managing this complexity, and thereby simplifying problem determination, lies in the study of *dependencies* between system hardware and software components. Much work is evident 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 requisite task of automatically obtaining accurate, detailed, up-to-date dependency models from a complex distributed system; most existing problem-determination work assumes the pre-existence of a manually-constructed dependency model, an optimistic assumption given the complexity and dynamics of many of today's enterprise and Internet-service systems. While there has been some work on automatically extracting static, single-node dependencies [8], there does not appear to be an existing solution for automatically detecting dynamic runtime dependencies, especially those that cross system and domain boundaries.

This paper addresses this deficiency via a new technique for automatically identifying and characterizing dynamic, cross-domain dependencies. Our technique differs considerably from traditional dependency-detection techniques by taking an *active* approach—explicitly and systematically perturbing system components while monitoring the system's response. The results of these perturbation experiments feed into a statistical model that is used to estimate dependency strengths. Compared to more traditional passive approaches based on knowledge discovery or learning algorithms, our active approach has the potential to obtain evidence of dependencies faster, more accurately, and with greater coverage. On the other hand, it is an invasive technique and therefore requires much greater care in how it is applied to production systems.

We have implemented our **Active Dependency Discovery (ADD)** technique and have applied the implementation to characterize a subset of the dependencies in a prototype e-commerce environment based upon the TPC-W web commerce benchmark, which simulates the behavior of an online bookseller's web storefront [11]. In particular, we used the ADD approach to generate a dependency graph for each of 14 distinct end-user interactions supported by the TPC-W environment; each such graph maps the dependencies between one user interaction and the particular database tables upon which that interaction depends. The results of these experiments reveal the power of the active approach: without relying on knowledge of the implementation details of the test system, our ADD technique correctly classified 139 of 140 potential dependencies and automatically characterized their relative importances (strengths).

This paper describes our active technique, its experimental verification, and our thoughts on how it can be used to assist in the root-cause-analysis phase of problem determination. We begin in Sections 2 and 3 with an overview of dependency models, their use in root-cause analysis, and related work. Next, Section 4 presents the details of our active technique for the discovery of dynamic cross-domain dependencies. In Section 5 we describe and discuss the results of our experimental validation of the

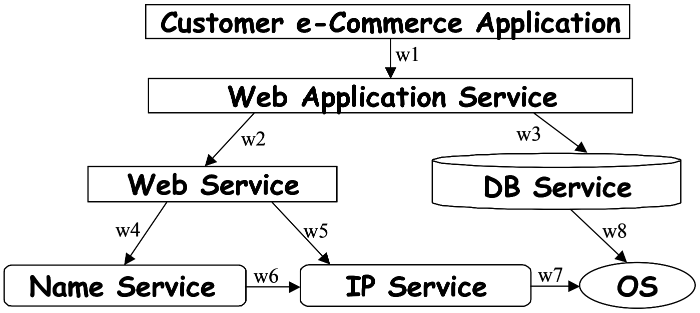

Figure 1: A sample dependency graph. Edge labels represent dependency strengths.

dependency-discovery technique in the context of the TPC-W web commerce environment, and we conclude with pointers for future work in Section 6.

2. Dependency Models

The basic premise underlying dependency models 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 is drawn between two nodes only if a failure or problem with the node at the head of the edge can affect the node at the tail of the edge; if present, the weight of the edge represents the impact of the failure's effects on the tail node. The dependency graph for a heavily simplified e-commerce environment is depicted in Figure 1.

Dependency graphs provide a straightforward way to identify possible root causes of an observed problem—one must simply trace the dependency edges from the problematic node (or entity) to discover all of the potential root causes. In the example of Figure 1, the dependency graph reveals that a performance degradation in the e-commerce application may be the result of a problem with the web service, which in turn may have been caused by a problem occurring within the name service. If weights are available on the dependency edges, as shown in the figure above, they provide a means of optimizing the graph search, as heavier edges represent more significant dependencies and therefore more likely root causes. We provide a detailed discussion of using dependency models for problem determination in Section 5.6, after first describing our active technique for obtaining such dependency models.

3. Related Work

There has been significant interest in the literature in using dependency models for problem diagnosis and root cause analysis. Two main approaches stand out. The first is in the context of event correlation systems, such as those described by Yemini et al. [12], Choi et al. [2], and Gruschke [4]. In these systems, incoming alarms or events are first mapped onto corresponding nodes of the dependency graph, then the dependencies from those nodes are examined to identify the set of nodes upon which the most alarm/event nodes depend. These nodes are likely to be the root causes of the observed alarms or events. The other main technique for using dependency models in root-cause analysis is to use the model graph as map for performing a systematic examination of the system in search of the root cause of a problem, as described by Kätker in the context of network fault management [7].

Most of these dependency-based root cause analysis techniques do not consider the details of how the required dependency models are obtained. We believe that for such techniques to be effective, they must be supplied with high-quality dependency models that reflect an accurate, up-to-date view of system state. Surprisingly, however, there is little existing work on the problem of automatically generating such high-quality dependency models, especially at system levels above the network layer and at the level of dynamic detail we believe necessary.

What little work there has been has focused on passive approaches to constructing dependency models. Kar et al. describe a technique for automatically extracting dependencies between software components within a given machine, based on data contained in existing software deployment repositories [6]. While this technique is effective for identifying static dependencies, it does not address the problem of obtaining dynamic, operational dependencies—dependencies that arise or are activated during the runtime operation of the system. It is important that these dependencies be modeled, as without them the overall dependency model reflects only a generic view of how the system might potentially be used, rather than how it is actually being used. Furthermore, the approach in [6] does not consider the issue of identifying dependencies that cross machine boundaries, a key component required for dependency models of realistic systems.

An interesting approach that does attempt to characterize both dynamic and cross-machine dependencies is described by Ensel [3]. This approach provides indirect evidence of dependencies and can detect dependencies that are exercised while monitoring is active. It cannot, however, provide evidence of causality, only of correlation, and thus cannot guarantee that the identified dependencies are real. In contrast, our active perturbation-based approach provides evidence of causality and can detect dependencies that rarely (or never) occur naturally during the monitoring period.

4. Detecting and Characterizing Operational Dependencies

4.1 Overview

Given the assistance that a detailed operational dependency model can provide to the task of root cause analysis, it is natural to consider how such models might be constructed or automatically extracted from a real system. There are two basic approaches that might be taken.

If the system is simple and its internal operation is well-understood, then a direct approach can suffice. That is, for each task that a system component can perform, a human expert can analytically compute the operational dependencies on other components. However, this approach quickly breaks down when the system grows more complex or when the source code and implementation details of the system are unknown. In such real-life situations, a more indirect approach to dependency discovery is needed, in particular one based on measurement and inference.

The essence of an indirect approach is to instrument the system and monitor its behavior under specific use cases as failures and degradations occur. Dependencies are revealed by correlating monitoring data and tracing the propagation of degradations and failures through the network of hardware and software components in the system. Dependency strengths can be calculated by measuring the impact on the dependent component of varying levels of degradation of the antecedent component.

The main challenges in any indirect approach are *causality* and *coverage*. Causality involves differentiating causal relationships indicating dependencies from simple correlations in monitoring data, whereas coverage implies collecting as much of the dependency model as possible, especially including portions that might be revealed only during faulty operation. There are several indirect approaches that can be considered, but most do not sufficiently address these two challenges, typically because they rely on various styles of postmortem time-correlation analysis performed on monitoring data covering only a subset of the possible failure states (a detailed discussion is beyond the scope of this paper). Thus, we choose to investigate a novel active-perturbation approach in which we explicitly inject problems into the system, monitor service behavior, and infer dynamic dependencies and their strengths by analyzing the monitored data. This approach solves both challenges: the controlled perturbation can be applied to every system component (providing full coverage), and the knowledge of which component is being perturbed disambiguates cause and effect (identifying causality). The following section explains our approach and sets up the background for the specific perturbation experiments that we have conducted.

4.2 Active Dependency Discovery

This idea of using explicit system perturbation to elucidate dependencies is the crux of a procedure that we denote **Active Dependency Discovery (ADD)**. The ADD procedure builds an operational dependency graph for a particular combination of system and workload while requiring very few details of the internal implementation of the system. The procedure consists of four major steps: node/component identification, system instrumentation, system perturbation, and dependency extraction.

Step 1: Identify the nodes in the operational dependency graph. In essence, this step boils down to enumerating the hardware and software components in the system, excluding only those components whose quality of service or potential for failure are irrelevant to the system. The information for this first step can come from a variety of sources: system deployment descriptions, inventory management systems like Tivoli Inventory [9], or from coarser-grained dependency models such as the automatically-generated structural models described by Kar et al. [6].

Step 2: Instrument the system. This involves establishing monitors for performance, availability, and any other relevant metrics. The instrumentation can be at the level of end-user-visible metrics, or can be placed throughout the various levels of the system.

Step 3: Apply active perturbation to the system in order to unveil its dependencies. The step begins by applying a workload to the system; the workload can either be a representative mix of what would be seen in production operation, or a targeted workload designed to explore dependencies corresponding to one component of the production workload. As the workload is applied, components of the system are perturbed at varying levels of intensity while the system instrumentation is used to record the system's behavior, performance, and availability.

A key decision to make when implementing the perturbation step lies in the selection of perturbation patterns, that is, what components should be perturbed and in what order. A good starting point is to systematically perturb every component in the system, one component at a time. If there exists some *a priori* knowledge of the structure of the dependency graph (for example, if the nodes were obtained from a static dependency model), then this graph can simply be traced from leaves to root to obtain a perturbation ordering; otherwise, the ordering may be arbitrary. More complex perturbation patterns involving multiple components can also be used to uncover dependencies on replicated or redundant components.

Step 4: Analyze perturbation data and extract dependency information. This is done with a combination of standard statistical modeling/regression techniques and simple graph operations. First, for each instrumented system component or metric, a statistical model is constructed that relates the measured values of that metric to the levels of perturbation of the various system components. These models are used to identify potential dependencies and to estimate their strengths: if the effect of a perturbation term is statistically significant, then we assume the existence of a dependency between the instrumented entity and the entity corresponding to the perturbation term; the value of the effect (the coefficient of the perturbation term in the model) is then taken as the dependency strength.

Given these statistical models, an operational dependency graph can be built by taking the set of nodes obtained in the first ADD step and adding directed edges corresponding to the statistically-significant dependencies identified by the models.

The active dependency discovery procedure provides a straightforward, easily-automated method of obtaining an operational dependency model for a given system. However, there are several issues that arise when considering practical deployment of ADD. We will consider the two most important here.

First, the ADD procedure is workload-specific, and produces dependency models that reflect the operation of the system under the workload used during the perturbation experiments. This can hinder use of the dependency model in problem determination if the workload present when the problem occurs does not match that used when constructing the model. One solution is to build a dependency model for each of the components of a system's workload, then select the most appropriate model based on the work in progress when a problem occurs. This is the approach we will take in the experiments of Section 5.

Second, and more importantly, the ADD procedure is invasive. Because ADD is based on perturbation, the procedure can noticeably impact the behavior, perfor-

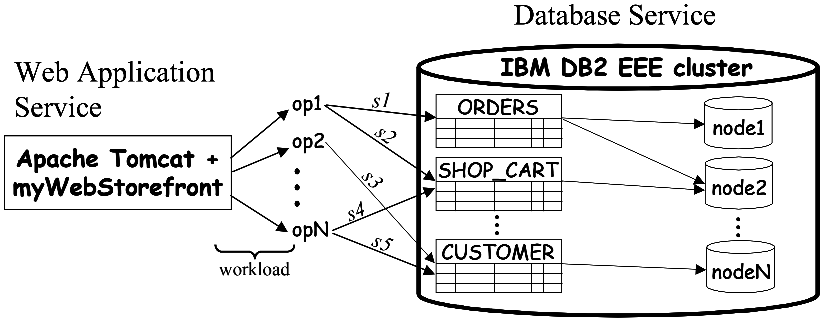

Figure 2: Operational dependencies between web application service, database service and internal database tables, broken down by workload component.

mance, and availability of the system while it is in progress. While this degradation is unacceptable in a production environment, it can be avoided or masked by running the perturbation as part of a special characterization period during initial system deployment, during scheduled downtime, or on a redundant/backup component during production use. Alternately, it may be possible to develop techniques for low-grade perturbation that could allow the ADD procedure to run (albeit slowly) during off-peak periods of production operation.

Finally, although we believe that the ADD procedure can be entirely automated, our description and implementation still rely on some manual intervention, in particular in designing and placing the measurement and perturbation points (although the statistical analysis to extract dependencies is fully automated). This is of particular concern for multi-tier systems, in which monitoring and perturbation may need to be placed at each tier-boundary in the system. Further research is needed to gauge whether an automated ADD system could use existing inter-tier interfaces for these purposes, or whether manual instrumentation will remain a requirement.

To illustrate the working of ADD we will use the example shown in Figure 2. Here the dependency edge represented by label *w3* in Figure 1 has been expanded to include the operational dependency edges (*i.e.*, those that come into play at runtime) between the web application service and the database service. We have also broken out the web application service into multiple nodes reflecting the different workload components (*e.g.*, user operations or transaction types) that could be applied to the system.* The goal of ADD is to (a) discover these operational dependencies and (b) estimate values of their strengths, denoted by *s1*, *s2*, etc. in Figure 2.

5. Experimental Validation of ADD

5.1 Overview

In order to validate ADD's effectiveness, we chose to implement the procedure in the context of a small but fully-functional web-based e-commerce environment. In partic-

*The overall dependency strength *w3* represents a weighted average of the strengths of the operational dependency edges (*s1*, *s2*, etc.), with the weights determined by the typical applied workload.

ular, following the specification in the industry-standard TPC-W web commerce benchmark, we built a three-tier testbed system implementing an on-line storefront application for a fictitious Internet bookseller. The goal of our experiments was to use ADD to identify and characterize operational dependencies in this environment, to be used as an aid in problem determination, in, for example, e-commerce systems.

The dependencies that we chose to investigate were those between the storefront service/application and individual tables in the back-end database, similar to the labeled dependencies in Figure 2. In particular, we built operational dependency models for each of fourteen different types of user interaction with the storefront service; the computed dependencies in each model indicated which database tables were needed to process the associated user request, and the strengths of those dependencies characterized the importance of each of those tables to the user request. This is a particularly appropriate set of dependencies to study for these first experiments, as the discovery problem is reasonably challenging, yet the results are easily validated by examining the application's source code.

5.2 Testbed environment

Our primary goal in constructing our testbed environment was to make it as realistic as possible given the constraints of our available hardware and software. A major requirement was therefore that the testbed implement a service or application that was as close as possible to one that might be deployed in real life. To address this requirement, we chose an application based on the specification supplied with the TPC-W web commerce benchmark. TPC-W is a respected industry-standard benchmark released by the Transaction Processing Performance Council, and is designed to simulate the operation of a realistic "business-oriented transactional web server" [11]. It includes both the specification for a fictitious Internet bookseller storefront application as well as a detailed specification for generating a reproducible user workload that is designed to be representative of actual user traffic patterns. Note that TPC does not supply an implementation of the TPC-W benchmark; we used a Java implementation developed by the University of Wisconsin, which included the application business logic, the workload generator, and a database population tool [1] [10].

The TPC-W storefront comes very close to our goal of deploying a realistic service. It includes all of the required components for an e-commerce application: a web interface, reasonably sophisticated business logic (including catalog searches, user-based product recommendations, "best-sellers", etc.), and a large back-end database.

Our TPC-W testbed system was organized in typical multi-tier fashion, with a web browser client tier, a web server front-end tier, a middleware tier, and a back-end database tier. The middle tier implemented the application's business logic via Java servlets deployed in a web application server [5]. The system was partitioned across three machines, with the web server and application server sharing a machine. The Wisconsin TPC-W implementation was installed on the system and configured with scale parameters of 10 000 items in the database and 50 expected simultaneous users. The 10 database tables and the web server's static image repository were populated with synthetic data according to the TPC-W specification.

5.3 Workload and perturbation

During all of our experiments, we applied the standard TPC-W “shopping” workload mix, designed to mimic the actions of typical Internet shoppers with a combination of roughly 80% browsing-type interactions and 20% ordering-type interactions. There are a total of 14 possible user interactions with the TPC-W environment, all of which were present in the workload mix, and as noted above our goal was to generate an operational dependency model for each of the 14 types of interaction.

The workload mix was applied using the Wisconsin-supplied Remote Browser Emulator (RBE), a threaded Java-based workload generator. We applied a load of 90 concurrent users; each simulated user carried out state-machine-based sessions with the server according to the distributions specified in the shopping mix. The server was not significantly saturated during the experiments. The workload generator recorded the start time and the response time for each simulated user interaction carried out during the experiments. User think time was simulated according to the specification.

A crucial part of our experiments was the perturbation of the system. As introduced above, our goal was to establish dependencies on the particular tables in the TPC-W back-end database, and as such we needed a way to perturb those tables. Our solution was to perturb the tables by introducing lock contention via a DB2 command that exclusively locks a particular table against any accesses by other transactions. This effectively denies access to the locked table, forcing other transactions and queries to wait until the table lock has been released and thereby perturbing their execution. We toggled the exclusive lock on and off during execution, with a full cycle time of 4 seconds and a duty cycle determined by the requested degree of perturbation. The table perturbation was managed by a Java client we developed that allows for multiple simultaneous perturbation of different database tables and for varying levels of perturbation over the course of the experiments.

5.4 Results

We carried out a sequence of 11 experiments to extract the dependencies in our testbed system. The first experiment characterized the normal behavior of the system: we applied the TPC-W shopping workload mix for 30 minutes and measured the response time of each transaction generated from the workload mix. The remaining 10 experiments investigated the effects of perturbation on each of the 10 tables in the TPC-W storefront’s back-end database. In each of these experiments, we applied the TPC-W shopping workload mix for two hours while perturbing one of the 10 database tables. For the first half-hour, the perturbation level was set at 25%; in the remaining three 30-minute periods, the perturbation levels were set at 50%, 75%, and 99%, respectively.

We begin our discussion of the results of these experiments by examining some data that illustrates the power of our perturbation technique. Figure 3 shows two graphs that plot the response times for one particular user transaction under different levels of perturbation for two database tables. We know *a priori* that this transaction depends on the ITEM and AUTHOR tables. The left-hand graph, Figure 3(a), shows the

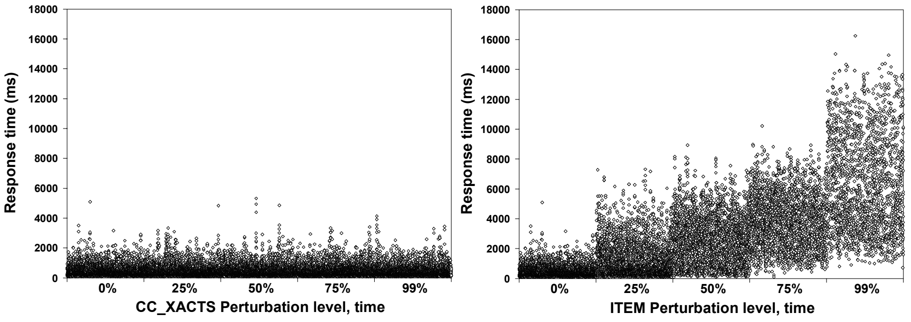

Figure 3: Raw response times for the TPC-W "execute search" transaction under various levels of perturbation. Within each perturbation level, values are plotted in increasing order of transaction start time.

response time for this transaction when an uninvolved table, CC_XACTS (holding credit card transaction data), is perturbed, whereas the right-hand graph, Figure 3(b), shows the response time behavior when the ITEM table is perturbed. Notice that there is a clear difference between the graphs: the left-hand graph displays no discernible indication that the response time varies with different perturbations of the uninvolved table, whereas the right-hand graph shows a clear shift in the response time distribution as the involved table (ITEM) is perturbed at increasing levels. This data directly suggests the presence and absence of dependencies for this transaction type: the evident shifts in distribution in the right-hand graph reveal an apparent dependency on the ITEM table, while the lack of such shifts in the left-hand graph excludes the possibility of a dependency on the CC_XACTS table.

5.5 Data analysis

While visual inspection of scatter plots such as those in Figure 3 is effective in detecting dependencies, it is not especially efficient, nor does it provide a numerical characterization of the dependency strength. We would prefer to be able to identify and measure dependencies automatically by applying statistical modeling techniques, as described below. However, there are some obstacles to overcome in performing the modeling, notably the distribution of the data and the sheer number of data points. As can be seen in Figure 3(b), the data distribution shifts from a clearly heavy-tailed distribution in the case of no perturbation to a more evenly distributed block under higher perturbation levels. The variance also increases significantly with the perturbation level. We addressed both of these problems (along with the sheer number of data points) by summarizing the data for each perturbation level as a simple mean of the logs of the original response times. This reduces the number of data points from approximately 14000 to 5 and makes the distribution more close to normal via the central limit theorem. As a side effect, it also appears to linearize the data quite well.

With the data thus reduced and linearized, it becomes easy to fit a regression line relating the mean log response time to the perturbation level. The regression line is the key to automatically identifying and characterizing dependencies: *a statistically*

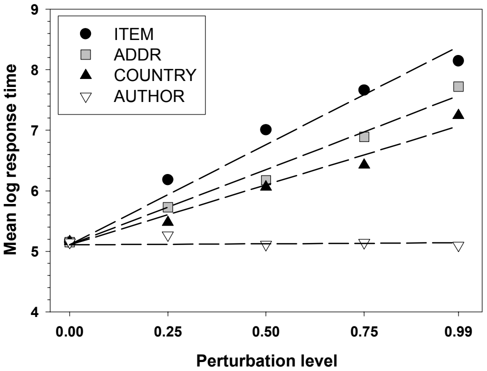

Figure 4: Mean log response times for the TPC-W "buy request" transaction for varying levels of perturbation of four database tables. Each of the four plots corresponds to the perturbation of a single database table. The dashed lines are regression lines obtained by fitting a first-order linear model of the form of (1) to the data.

nonzero slope for this line indicates the existence of a dependency, and the magnitude of the slope characterizes its strength. More formally, consider a first-order linear regression model of the form:

$$r_{ik} = \mu_i + \sum_{j=1}^{10} \alpha_{ij} P_j + \varepsilon_{ik}, \ \ \varepsilon_{ik} \sim N(0, \sigma_i) \tag{1}$$

where i indexes the transaction type $(1, \ldots, 14)$, r_{ik} is the value of the mean log response time for the k'th execution of the i'th transaction type, j indexes the 10 database tables, P_j is the level of perturbation of table j, μ_i is the mean log response time in the unperturbed state, and α_{ij} is the *effect* of the perturbation of table j on the mean log response time for transaction i, and corresponds to the slope of the regression line.

If such a model fits the measured data well, then we can use the α_{ij} terms as direct measures of the dependencies of transaction type i on table j. Specifically, if α_{ij} is statistically non-zero (at, say, the 95% confidence level), then we conclude that transaction type i depends on table j with a strength of α_{ij}.

As an example, let us consider the data collected for a different user transaction, in this case the "buy request" transaction. A user executes this transaction by clicking the "checkout" button, indicating that they are ready to purchase the contents of their virtual shopping cart. Figure 4 plots the mean log response times for this transaction for four cases, corresponding to the perturbation of four different tables: ITEM, ADDRESS, COUNTRY, and AUTHOR. It also plots the fitted model results as dashed lines. For reference, the correlation coefficient for the fit was $R^2 = 0.983$. As can be seen in the figure, there are clear differences in slopes between the various lines. One line (corresponding to the AUTHOR table) has a slope that is very close to zero—in this case, we assume that there is no dependency here on the AUTHOR table. The other three lines, however, have positive slopes (3.31, 2.49, and 1.98, respectively), indicating dependencies. From the differences in slope, we draw the conclusion that the

	AdmConf	AdmReq	BestSell	BuyConf	BuyReq	CustReg	Home	NewProd	OrdrDisp	OrdrInq	ProdDet	SrchReq	ExecSrch	ShopCart
Address				+	#				#					
Author	+	*	*					#			**X**		#	
Cc_Xacts				+					#					
Country				+	*				*					
Customer				*	#	#			#					
Item	*	#	*	#	**X**		#	#	#		**X**	**X**	#	**X**
OrderLine	--		+	*					#					
Orders	*		*	#					**X**					
ShopCart														#
ShopCartLine				*	#									#

Figure 5: Summary of operational dependencies discovered by applying ADD to the TPC-W system. The 14 TPC-W transaction types are listed across the top, the 10 TPC-W database tables down the side. A non-empty box at the intersection of a transaction and table indicates that the transaction depends on the table. The symbols represent the dependency strengths: **X** ∈ (3,4], # ∈ (2,3], * ∈ (1,2], + ∈ (0,1].

dependencies have different strengths, with the Item table having the strongest dependency, followed by the Address table and the Country table, in order.

If we consider all of the transaction types and tables defined by TPC-W, there are 140 *potential* dependencies—what might be claimed as dependencies based on a static analysis of the system. Referring back to Figure 2, these potential dependencies would be obtained by adding edges from each workload component of the application server to every table in the database. It is unlikely that all of these dependency edges would be present in practice, and in fact in our TPC-W case study, our application of the ADD technique identified only 41 of the potential 140 as true dependencies.

To verify this result, we carried out a detailed (and time-consuming) manual analysis of the source code of the TPC-W business logic to directly determine the true dependencies. From this manual analysis, we found that there were a total of 42 true dependencies in the system, of which the ADD procedure correctly identified 41, with the one misclassification due only to insufficient monitoring data. Thus we can conclude that in this case the ADD procedure performed remarkably well, correctly classifying 139 of 140 potential dependencies in an easily-automated procedure.

5.6 Application of ADD to Problem Determination

Figure 5 depicts in tabular form the operational dependencies discovered by applying ADD to the TPC-W system. The information content in the figure is identical to what would be included in a standard graph representation of the calculated operational dependency models. Visualizing the dependencies in this manner is suggestive of how useful operational dependency models might be for assisting in root-cause analysis.

For example, say that the system manager for this fictitious e-commerce storefront receives a report from a client that the system is not performing correctly (this report might take the form of a service-level agreement violation report). He or she can then take the following steps to narrow down the root cause:

1. Identify the faulty transaction, from the problem report or via test transactions.
2. Find the appropriate column in a table like that of Figure 5.
3. Select the rows representing dependencies. This gives the set of potential root causes and their weights
4. Investigate the potential root causes, starting with those with highest weight.

Unfortunately, this procedure may not uniquely identify the root cause of a problem. However, notice that the tabular representation of dependency data in Figure 5 is suggestive of a matrix. Consider a "basis" for the dependency space—a set of user transactions that, taken in different combinations, provides a way to isolate the effects of specific tables. For example, in Figure 5, notice that if we "subtract" the ORDERDISP dependencies from the BUYCONF dependencies, we are left with just a dependency on the SHOPCARTLINE table. If we could find sets of transactions that gave us a basis for all tables, we could use performance results from the basis set of transactions to uniquely isolate the faulty table. For example, the case above, if we knew that BUYCONF transactions were slow, but ORDERDISP transactions were running normally, we would know immediately that the problem was in the SHOPCARTLINE table.

In our TPC-W example, such a basis does not exist. However, as future work, we plan on extending the application with some extra business logic to define some synthetic transactions that would complete the basis set. This would provide us the extra power needed to do automatic root-cause analysis for the portion of the system modeled in the dependency graphs. When extended to a complete system model (rather than restricted to database tables as in our experiments), automated dependency analysis based on this extension to our ADD approach could offer a significant improvement to the state-of-the-art in problem determination.

6. Conclusions and Future Work

We have introduced active dependency discovery (ADD), a novel technique for determining dynamic, operational dependencies between services and components in distributed system environments, and have demonstrated its effectiveness in a small-scale but realistic case study. ADD differs significantly from most existing techniques in this area in that it relies on active perturbation of the system. While this is an invasive technique, it is also quite powerful, accurately revealing dependencies between components, characterizing their strengths, and conclusively identifying causality.

An efficient and accurate procedure for discovering a distributed system's operational dependency graph is a key component of several proposed problem determination and root-cause-analysis techniques, which are themselves key components of the system management problem. ADD provides such a procedure, and furthermore has the potential of improving on existing root-cause-analysis techniques by providing more accurate and complete dependency information, providing dependency strength information to optimize the search for root causes, and by forming the foundation for potential extensions that use our notion of an operation "basis" to automatically and precisely isolate root causes of observed problems.

However, the work described in this paper is only an initial step in probing the potential power and applicability of the ADD technique. Before ADD can be practically deployed, a host of open research issues must be resolved. Most importantly, techniques must be developed for integrating ADD's perturbation into production use in a non-disruptive manner, potentially via low-grade perturbation techniques or hybrid active/passive approaches. ADD must also be verified and extended for more complex e-commerce environments than TPC-W; in particular, the tradeoffs between end-to-end and layer-by-layer application of ADD must be examined, especially in systems using shared queuing to decouple their tiers. Finally, further research is needed to determine how to best use ADD-discovered dependency models for management tasks; promising directions include automated model-based root-cause analysis, and even the use of dependency strengths for system performance optimization.

References

[1] T. Bezenek, T. Cain et al., "Characterizing a Java Implementation of TPC-W," in *Third Workshop on Computer Architecture Evaluation Using Commercial Workloads*, Toulouse, France, 2000.

[2] J. Choi, M. Choi, and S. Lee, "An Alarm Correlation and Fault Identification Scheme Based on OSI Managed Object Classes," in *1999 IEEE International Conference on Communications*, Vancouver, BC, Canada, 1999, pp. 1547–51.

[3] C. Ensel, "Automated Generation of Dependency Models for Service Management," in *Workshop of the OpenView University Assoc.*, Bologna, Italy, 1999.

[4] B. Gruschke, "Integrated Event Management: Event Correlation Using Dependency Graphs, in *Proceedings of 9th IFIP/IEEE International Workshop on Distributed Systems Operation & Management (DSOM '98)*, 1998.

[5] The Apache Jakarta Project, http://jakarta.apache.org.

[6] G. Kar, A. Keller, and S. Calo, "Managing Application Services over Service Provider Networks: Architecture and Dependency Analysis," in *Proceedings of the Seventh IEEE/IFIP Network Operations and Management Symposium (NOMS 2000)*, Honolulu, HI, 2000.

[7] S. Kätker and M. Paterok, "Fault Isolation and Event Correlation for Integrated Fault Management," in *Fifth IFIP/IEEE International Symposium on Integrated Network Management (IM V)*, San Diego, CA, 1997, 583–596.

[8] A. Keller and G. Kar, "Dynamic Dependencies in Application Service Management," in *2000 International Conference on Parallel and Distributed Processing Techniques and Applications*, Las Vegas, NV, 2000.

[9] Tivoli, Inc., "Tivoli Inventory," http://www.tivoli.com/products/index/inventory/.

[10] "TPC-W in Java," http://www.ece.wisc.edu/~pharm/tpcw.shmtl.

[11] TPC Benchmark W Specification v1.0.1, Transaction Processing Performance Council, San Jose, CA, 2000, http://www.tpc.org/wspec.html.

[12] S. Yemini, S. Kliger et al., "High Speed and Robust Event Correlation," *IEEE Communications Magazine*, vol. 34, no. (5), pp. 82–90, May 1996.