Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination

Manish Gupta[1], Anindya Neogi[1], Manoj K. Agarwal[1], and Gautam Kar[2]

[1]IBM India Research Lab., New Delhi
{gmanish,anindya_neogi,manojkag}@in.ibm.com
[2]IBM Watson Research Center, New York
gkar@us.ibm.com

Abstract. In order to reduce mean time to recovery (MTTR) in heterogeneous enterprise environments it should be possible to easily and quickly determine the root cause of a problem detected at a higher level, e.g. through response time violation of a transaction category, and resolve it. Many problem determination applications use a component dependency graph to pinpoint the root cause. However, such graphs are often manually constructed. This paper introduces a simple non-intrusive technique based on mining of existing runtime monitored data, to construct a dynamic dependency graph between the components of an enterprise environment. The graph is traversed to identify nodes that are the cause of response time related problems.

1 Introduction

Typically dependency models of system hardware and software are analyzed for problem determination and impact analysis in complex enterprise environments. Prior work talks about explicit middleware instrumentation [5], or internal instrumentation of the components (via ARM [2]) for obtaining system dependencies. These methods are time consuming and are difficult to apply in legacy environments. The main contribution of this paper is in showing how existing performance monitoring infrastructure available in middleware, such as web application servers and database servers, can be used in discovering dependencies between the various components of a system. Management clients can poll the middleware for performance metrics, such as total number of requests to a component, average response time of a component, etc. This paper proposes a data-mining algorithm that uses this performance data for obtaining "probabilistic" dependencies between components. An online algorithm for discovering and updating these dependencies between components is provided. Because of the probabilistic nature of these dependencies, "false" dependencies may arise and therefore we show how a problem determination application can use the dependencies effectively. Dependencies can be of various types [17], but this paper focuses on finding runtime software/service dependencies among the following components: URLs, servlets, EJBs, and SQLs in a web application.

M. Brunner and A. Keller (Eds.): DSOM 2003, LNCS 2867, pp. 221-233, 2003.
© IFIP International Federation for Information Processing 2003

The rest of the paper is organized as follows. Section 2 compares our work with related research in dependency extraction. Section 3 presents the overall prototype system setup for dependency extraction, storage, and usage by management apps. In Section 4, we provide an algorithm for computing dependencies from the management data obtained from the managed resources and show how to use these dependencies for problem determination. Section 5 provides experimental validation of the algorithm on our testbed. Section 6 concludes the paper.

2 Related Work

A dependency graph of a system may be obtained 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. However, it is unsuitable to apply such methods in large and heterogeneous systems because they are system specific and do not provide runtime dependency information. Indirect methods operate at runtime, and may be intrusive, semi-intrusive, or non-intrusive with respect to the operational system in the manner they extract dependencies.

An example of an intrusive technique is one that relies on code instrumentation such as ARM [2]. Dependencies are calculated by correlating data gathered during the flow of transactions through various components. eWLM [1] is a workload manager that uses ARM to instrument the underlying components and extract a component dependency graph. PinPoint [5] is a problem determination framework, where coarse-grained client requests are tagged as they travel through and discover the components of an enterprise system. Tagging requires middleware-level instrumentation to pass the request ID between components, similar to ARM. There are certain other approaches [12], [15], and [18] that use instrumentation of application to get dependencies. They instrument the application code such that some probes or hooks are available to management application to get the data out of managed object and to exert control over the managed object.

A key problem with the above approaches is that they may be unusable in situations where multi-vendor components are used and in places where even transaction correlation code cannot be inserted into the system for security, licensing, or other technical constraints. Unless all components adhere to standards, such as ARM, instrumentation based approaches cannot be deployed on a large scale. This motivates the requirement of semi or non-intrusive approaches. An example of a semi-intrusive approach is Active Dependency Discovery [4], where perturbation/fault-injection is used to obtain dependencies.

In addition, Ensel [8] has also suggested using Neural Networks technology to automatically generate dynamic and cross-machine dependency graphs while monitoring is active. The technique however does not provide any evidence of causality and only detects correlation. However, at the time of this writing, there are no details available regarding the training of such networks and experimental or theoretical analysis of the accuracy and precision of the method. Steinder et al. [19] have also used the concept of belief networks for fault localization in network services built on complex communication topologies. The technique is specific to network services and the bipartite graph is developed specifically for problem determination.

Our approach falls in the non-intrusive category because we do not instrument any application and use whatever instrumentation is provided by the vendor of the middleware for generating dependencies between various monitored resources. We rely on the fact that most vendors provide some built-in instrumentation for monitoring statistics primarily for accounting and performance tuning purposes. The statistics, at a minimum, include invocation and average execution time counters for the monitored resources. In Section 4 we will see that, in order to compute a probabilistic dependency between any two monitored resources, knowing at least the above two statistics is sufficient. Also in Section 4 we will see that our technique is independent of the actual "type" of the resource such as servlets, EJBs, SQLs, and database tables and therefore can be applied to other resources.

There are several papers that talk about problem determination and root cause analysis using dependency graphs [2][6][8][10][16][22]. Yemini et.al. [22], Choi et.al. [6], and Gruschke [10] assume the existence of a dependency graph and show how incoming alarms and events may be mapped to nodes of the graph and how the graph may be used to identify dependent nodes, which are the likely root cause of problems. Katker [16] also uses the graph for systematic analysis of a problem to identify the root cause in the network fault management domain. Once we obtain the basic probabilistic dependency graph with potentially large number of edges, we propose a technique to traverse it to quickly identify the root cause of response time related problems in generic systems.

Hellerstein and Ma [13] have applied data-mining algorithms for discovering useful patterns in historical system event data. They discuss how data-mining can be used to identify actionable patterns and in particular they present algorithms for three kinds of frequently occurring patterns in event data. Thoenen et al. [20] have developed an event management and design methodology that has been widely used. The core of this methodology is a graphic representation of the roles and dependencies or relationships between events.

3 System Architecture and Design Issues

We have built a prototype system to demonstrate the effectiveness of the algorithm proposed in this paper. The prototype system development is guided by two principles: reuse of existing monitoring information and separation of management interface from the business interface. The first principle is motivated from the fact that most of the existing enterprise code (including legacy code) is not ARM-instrumented and its owners may be averse to introducing instrumentation code in their applications as well. But the middleware that runs this code may provide performance data that can be used to "guess" dependencies between the various components. Furthermore, our technique can come handy to obtain dependencies between components running on middleware from different vendors. The second principle is motivated by the general trend [9] in the industry to separate the management interface from the business one and DMTF's CIM [7] is one such popular effort to achieve this.

The system consists of three tiers as shown in Figure 1. The bottom layer is the managed system, which currently consists of WAS and DB2 running a TPC-W bookstore application. Both the WAS and DB2 have a monitoring API through which run-

time statistics (such as performance-monitoring counters) can be extracted by a local agents A1 and A2, respectively, and sent to a correlation engine (CE) in the central management layer. The data-mining algorithm implemented in CE takes this monitoring data to compute the dependencies between the monitored components. The topmost layer consists of management applications that pull raw dependency data from the repository through CIMOM for various uses. In our test-bed, the application and the database servers run on the same machine and hence both A1 and A2 send data that is time stamped using the same clock. In our next prototype version we will run a distributed system where the issues of clock synchronization will be handled.

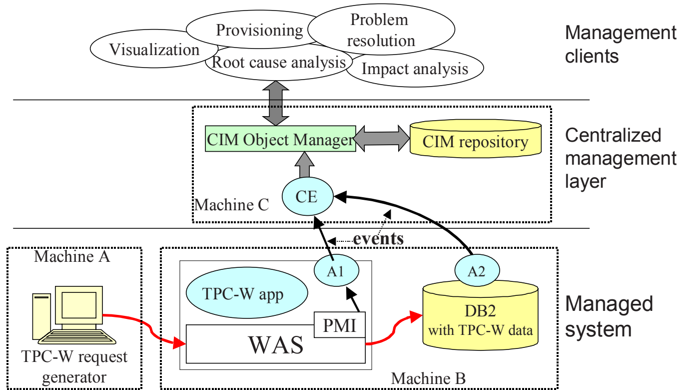


Fig. 1. Prototype System Setup: The IBM WAS v4.0 and IBM DB2 UDB v7.1 are installed on machine B (**2GHz, 1GB**). WAS v4.0 runs the TPC-W (see [21]) bookstore application consisting of 14 servlets and 46 SQLs and a database of 10,000 books. Machine A (**600MHz, 512MB**) is used to run the remote browser emulator (RBE) or the TPC-W request generator and sends URL requests (consisting of 50% buy and 50% browse) to machine B, which also runs the IBM HTTP server. On Machine C (**2GHz, 1GB**) runs the CIMOM and the CE. All Machines are connected over a 100 Mbps Ethernet

The current prototype monitors only a small subset of the objects/resources pertaining to WAS and DB2, namely, the URLs, servlets, EJBs, and SQLs[1]. In other words, the agent A1 provides events for URLs, servlets, and EJBs only, and A2 provides the events for SQLs. Each of the URLs, servlets, and EJBs are modeled using the CIM schema in the J2EE Management Specification [14]. The SQLs are modeled using the *CIM_UnitOfWorkDefinition* class in the CIM Metrics Model [7]. In order to capture the dependency information between the above objects, as for example the dependency between a servlet and an EJB, we introduced additional association classes, which derived from *CIM_dependency*. As and when the instances of the CIM

[1] In an enterprise system the web transactions (or URLs) are typically serviced by servlets which in turn may invoke EJBs or directly execute SQLs. The TPC-W bookstore application [21], on which we have tested our technique, comprises a collection of servlets each of which issues SQLs directly to the database. Also, the WAS' and DB2's monitoring API allows us to obtain the performance metrics that are sufficient for computing the probabilistic dependencies.

classes pertaining to the URLs, servlets, EJBs, SQLs, and their dependencies are discovered, the CIM repository in the middle layer in Fig. 1 is populated with them.

4 Dependency Graph Extraction and Its Usability

This section presents the algorithm to extract dependency relationships from existing monitoring data. It also describes how the algorithm may produce false dependencies and the manner in which a problem determination application may cope with them.

Dependency Definitions. Consider any two components, say *A* and *B*, where *A*, for example, may be a servlet, and *B* an EJB. 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. 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. Any dependency between *A* and *B* thus arises from an invocation of *B* from *A*, which may be synchronous or asynchronous. Note that the algorithm described in this paper handles synchronous invocations only.

Corresponding to each servlet, EJB, or SQL request an event, carrying the essential statistics such as the number of requests and average response time *so far* to the component, is received by the CE (in Fig. 1). Associated with each such event is an activity period, which for the purposes of the discussion here can be taken to be a time interval with its start time and end time being the start time and end time, respectively, of the request to the component represented by the event (see [11] for more detail). We say that an activity period $[b_1, b_2]$ of the component *B* is *contained* in an activity period $[a_1, a_2]$ of the component *A* if $a_1 \leq b_1$ and $b_2 \leq a_2$. Finally, our definition of dependency between two the components *A* and *B* is as follows: we say that *A depends on B with strength p* (or $A \xrightarrow{p} B$ [2]) if the probability that a given activity period of A contains an activity period of B is *p*.

The above definition of containment of an activity period into another captures the following dependency type that we are interested in.

- *A* invokes the services of *B*, *directly* or *indirectly*, and *A* finishes only after the invocation of *B* returns. An example of *A* invoking *B* directly is: *A* calls a method of *B* and waits for the call to return. On the other hand, we say *A* invokes *B* indirectly if *A* directly invokes some component *C*, which in turn can either directly or indirectly (a recursive definition) invoke *B*.

A dependency of the above type is what we call a *true* dependency, and any other type of dependency that is captured by our dependency definition is what we term as a *false* dependency. The motivation for choosing the definition for *p* given above is that we believe that the number *p* comes very close to the probability that a given execution of *A* invokes, directly or indirectly and at least once, the component *B*, and fin-

---

[2] For expositional simplicity we occasionally omit the strength value on the arrow and simply say A→B.

ishes only *after* the invocation to B returns. If A calls B's methods on each request to A then, as per our definition, p is 1.0. If only 20% of the requests to A result in calls to B then p is 0.2. Note that multiple calls or containment of activity periods of B per request to A are counted as a single call to B. In Fig. 2 in [11] we provide another example to describe this concept of containment that not only captures the notion of true dependencies but also of false ones.

In the following subsection, we present an algorithm for computing the dependencies (based on our definition above) for any given component, say A. The true dependencies for the component A include all the components that are directly and indirectly invoked by A. Furthermore, we expect that this algorithm, in the process, also computes false dependencies. A natural *two-level* dependency graph, resulting out of the above definition of a dependency, has a node at level one with all its true and false dependencies situated at the second level. Fig. 3 in [11] shows a two-level dependency graph. Looking at only the two-level graph of a component suffices because all the monitored components that *are* dependencies of the component are captured at the second level of the graph and there is no need to traverse the two-level graphs of any of the dependencies (i.e., the two-level graphs of the antecedent nodes). Furthermore, as the transitive property (see [11]) of dependencies may not hold in general, a two-level graph helps us to consider *only* the true dependencies of a component.

The number 'p' in $A \xrightarrow{p} B$ (hereafter, referred to as the p-value) depends on the business logic in A and the workload applied to the enterprise environment containing the component A. As the logic and the workload change so will the p-value. Assuming that the logic and the workload do not vary with time, we now show how CE (see Fig. 1) estimates p-value from the event traces. From the definition of p-value given above, the straightforward way of estimating it is to calculate the fraction of the number of activity periods of A that contain some activity period of B from the event traces received for both A and B. That is, let $\#A$ denote the total number of activity periods of A seen so far, and $\#(B, A)$ denote the total number of activity periods out of $\#A$ that contain at least one activity period of B. Then the number $\#(B, A)/\#A$ is an estimate of the p-value. It is desirable that as new events are received at the CIM repository (see Fig. 1) our algorithm (to be given in the next subsection) generates and updates the dependency graph online.

Online Dependency Extraction Algorithm. We will present the algorithm informally. Consider only a pair of event types; say servlet and SQL events, for expositional simplicity of the algorithm. Let Σ_{sql} denote the set of all SQLs and $\Sigma_{servlet}$ denote the set of all servlets in a given web application. Let $A \in \Sigma_{servlet}$. The goal is to discover and update all the dependencies $A \xrightarrow{p} B$ where $B \in \Sigma_{sql}$. A key property that our algorithm uses that satisfied by the system in Fig. 1 is that the events from a given component (say servlet A) are received at CE (in Fig. 1) in the increasing order of their time stamps where the time stamp of an event is defined as the end time of the activity period represented by the event (see Section 4 in [11]). This also implies that SQL events received at CE are in the increasing order of their time stamps.

A few definitions are in order before we present the algorithm. We say that a servlet event is *fully processed* if all the SQL events that are contained[3] in the servlet event have been identified; otherwise we say that the servlet event is *partially processed*. The following event lists are maintained: *antecedentList* comprises events received for SQLs in Σ_{sql}, *dependentList* comprises events received for the servlet A, and *dependencyList* maintains the list of dependencies, of A, obtained so far. The *dependencyList* variable, essentially, keeps the list of all those SQLs in Σ_{sql} for which dependencies have been detected so far. We define *LPP* as an event pointer that at any time holds reference to the currently lowest time stamped partially processed event in *dependentList*. We also define *HSQ* as an event pointer that at any time holds reference to the currently highest time stamped event in *antecedentList* subject to its time stamp being less than or equal to the time stamp of the servlet event referenced by *LPP*. A counter *#A* (initialized to zero) keeps the count of the number of fully processed servlet A events so far. For each SQL B in the *dependencyList* we set a counter #(B, A) (initialized to 1) that counts the number of servlet A events so far that contain at least one SQL B event. For any event pointer *e* the notation *e*.id, *e*.timestamp, *e*.starttime, *e*.next, and *e*.prev, respectively, refer to the following attributes of the event referenced by *e* in an event list, viz., the identifier, the time stamp, start time, reference to the event immediately *after* the event referenced by *e*, and reference to the event immediately *before* the event referenced by *e*. Both *LPP* and *HSQ* are initialized to null (see [11] for a complete description).

Dependency Algorithm (an outline)**:** The main algorithm preserves the following property at all times: all servlet events in the *dependentList* having time stamps less than *LPP*.timestamp are fully processed and the rest are partially processed, and *HSQ* always points to the highest time stamped SQL event, currently in the *antecedentList*, having timestamp less than or equal to *LPP*.timestamp. In the non-boundary case, i.e., when both *LPP* and *HSQ* are non-*null*, on an arrival of an SQL event with time stamp less than *LPP*.timestamp we set *HSQ* to this event, otherwise *LPP* is set to *LPP*.next, i.e., to the *next* servlet event, whenever it becomes available, and the count #A is incremented by one. In the latter case, we continue to scan the SQL events to the left of *HSQ* until we reach an SQL event whose end time is less than the start time of the new servlet event pointed to by *LPP*; thereafter we scan the SQL events on the right to *HSQ* and compute the new value for *HSQ*. Each time we compare an SQL event with the servlet event pointed to by the *LPP* we also check for containment of the activity period of the SQL event into the servlet event, updating the *dependencyList* and corresponding frequency counts if required. This process is repeated with the new *LPP* and *HSQ*. The algorithm outputs estimates of *p*-values by computing for each SQL B in the *dependencyList* the number #(B, A)/#A.

It is easy to see that the algorithm has the desirable property that the number of event comparisons needed in order to ascertain all the events contained in another event is minimum. In other words, if n is the number of activity periods contained in a given activity period then the algorithm finds that out in $O(n)$.

---

[3] An event is *contained* in another event if the former's activity period is contained in the latter's.

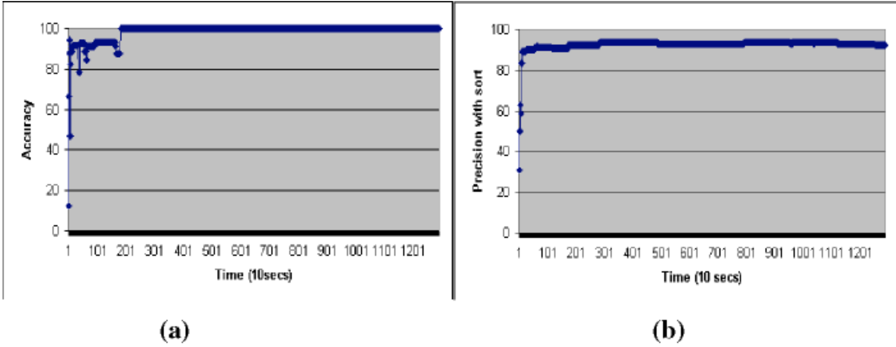
Problem Determination (PD) Using Probabilistic Graph. The presence of false dependency edges in a two-level graph complicates the traversal order of edges in the two-level graph of a component. Note that the edge weights (or strength), in a sense, correspond to the "likelihood" of *A* depending on *B*. False dependencies may be caused by false containment generated due to concurrency of transactions flowing through the server components. Clearly, traversal of false dependencies reduces the performance of the PD algorithm and increases the MTTR.

We introduce another statistic of a dependency $A \rightarrow B$ called, *r*-value, which we use along with the *p*-value to achieve an improved ordering of the dependencies of a node *A*. We define *r*-value as the probability that a given activity period of the component *B* is *contained in* an activity period of the component *A*, where the definition of containment is as described earlier. We estimate *r*-value as follows. Let, $\varphi(A, B)$ denote the number of activity periods of *B*, seen so far, which were found to be contained in at least *one* activity period of *A*. Finally, let $\#B$ be the total number of activity periods of *B* seen so far. We take the number $\varphi(A, B)/\#B$ as an estimate of *r*-value. To see the difference between *p*-value and *r*-value, consider the earlier example in which *A* invokes *B* only 5% of the time *A* is active. Suppose *A* is the *only* component that invokes *B* in the application, then the *p*-value will be close to 0.05 whereas the *r*-value will be close to 1.0. Another illustration of the difference between *p* and *r*-values is given in Table 1 in [11]. The algorithm for computing *p*-values given earlier an be easily tailored to compute the *r*-values as well.

We now introduce a heuristic that uses the two statistics, the *p*-value and the *r*-value, to order the dependencies of a node: we sort the dependencies of a node *A* in the *non-increasing* order of $\mathbf{max}(p, r) + pr$. The rationale for choosing the heuristic is the following. The first term coarsely sorts the dependencies of a node so that edges with high *p*- or high *r*-values are catapulted up in the order. A dependency is more likely to be true if at least one of these is high. The product term performs finer grain sorting among equals. A dependency is more likely to be true if both *p*- and *r*-values are high compared to the case when only one of the values is high. In this heuristic the true dependencies, which have low *p- and r*-values, will be penalized. For example, if a servlet *X* executes an SQL *Y rarely*, but the SQL *Y* is *frequently* executed by some other servlets in an application, then the dependency *Y* of *X* will appear lower down in the sorted list of dependencies of *X*, because both *p* and *r* values will be small. However, there is an increased likelihood that if the problem in *X* is due to *Y*, then other servlets that more frequently call *Y* also have the same problem and we are able to identify *Y* through one these servlets.

5 Experimental Evaluation

This section presents an experimental evaluation of the dependency algorithm described in Section 4 on the testbed, which consists of the three machines as described in Figure 1.



(a) **(b)**

Fig. 2. The variation in (a) Accuracy and (b) Mean Precision with time for a particular load of 50 customers. Accuracy and precision values stabilize as the entire graph is discovered. They remain stable with the workload, thus the computed graph can be used for long stable periods. It can be observed that the method is 100% accurate (Fig. 2 (a)) as expected, however precision stabilizes to around 98% (Fig. 2 (b)). The convergence time is also within a few minutes and closely tracks workload stabilization

Definition of Accuracy and Precision. The aim of the experiments is to discover the dependencies between servlets and SQLs. In our testbed, there are 54 true servlet-to-SQL dependencies out of the potential set of 644 dependencies. So if we discover all 54 true dependencies then the accuracy is 100%. *Accuracy* is defined as the percentage of the true dependencies discovered. *Precision* is defined based on the two-level graph traversal order in Section 4. In the sorted dependency list of a node having n edges, the edges are numbered (starting from the first edge) from 1 to n with m ($\leq n$) being the last true dependency in the list. We assign a weight of m-i to the dependency labeled i, where $1 \leq i \leq m$-1. The sum of the total weights from 1 to m-1 is therefore $w_{tot} := m(m-1)/2$. Out of w_{tot} the total contribution of weight coming from false dependencies is defined as $w_f := \sum_{i<m,\, i \text{ is a false Dependency}} (m-i)$. Finally, we define percentage *node-precision* as $100\,(1-\frac{w_f}{w_{tot}})$. Observe that this definition penalizes a false dependency more if it occurs higher in the list. The precision value reported in the following experiments is the *mean* percentage node-precision over the 14 servlets.

Accuracy and precision are measured on the testbed as follows. We have instrumented the TPC-W application with an in-house developed transaction correlation code to find the actual set of the dependencies that should be discovered at each point in time as user transactions flow through the system. Thus accuracy and precision can be computed at each time point by comparing the dependency information generated by our dependency algorithm with the accurate information generated by instrumented transaction correlation code in the same experiment. For illustration see Fig. 4 for an experiment run with 50 simultaneous customers.

Table 1. Precision and Accuracy with Load for traffic mix: 50% buy and 50% browse. The WAS is configured to fork threads on demand with a minimum of 25 pre-forked threads. As customer load is increased, the number of simultaneously active threads grows, and the thread pool size is also automatically increased by the system beyond 25, if needed. The number of customers is increased till a significant percentage of URL requests timed out due to load. The WAS machine can support up to 200 customers. Hence, the experiments have been run only up to 200 customers as a high load case. Each experiment was run for one-hour duration. If we run them for longer duration, we expect precision values to go up

Load: #Customers	#Avg active threads (Avg thread pool size)	Mean Precision % (std dev σ)	Accuracy %
5	2.2 (25.0)	100 (0)	100
25	3.5 (25.0)	99.94 (0.22)	100
50	4.6 (25.0)	98.33 (3.12)	100
100	10.6 (25.0)	81.86 (21.00)	100
150	17.1 (26.4)	71.52 (32.43)	100
200	20.0 (31.9)	62.62 (26.82)	100

Table 2. Effect of polling rate on precision and accuracy with a fixed load of 100 customers. Precision is more sensitive to polling rate than accuracy. At lower polling rates, even though accuracy is 100% and the management overhead drops, the precision also goes down. At higher load, high polling rate is required to maintain high precision, however it also increases the overhead, which is undesirable at high load

Polling interval in millisecs	Mean Precision % (std dev σ)	Accuracy %
50	88.71 (18.75)	100
100	81.86 (21.00)	100
500	77.03 (29.82)	100

Table 3. Overhead Measurement in terms of percentage increase in throughput and response time by varying polling interval and customer load. Each reading corresponds to an experiment that was run for 3 hours

Simultaneous Customers (load)	Polling Interval (milliseconds)			
	100		500	
	Throughput (%)	Response Time (%)	Throughput (%)	Response Time (%)
10	2	10	0	3
200	9	33	8	25

Accuracy and Precision for TPC-W Bookstore Application. As more multiple transactions proceed simultaneously, it is harder to separate true dependencies from false ones using containment relationship. The degree of concurrency of transactions on WAS may be increased by increasing the number of simultaneous customers or browser emulators, thus keeping more threads in the thread pool simultaneously active. Table 1 shows the variation of accuracy and precision values with increasing customer load and concurrency. The algorithm shows 100% accuracy under all loads. Precision values decrease with increasing load. The standard deviation (σ) for precision is low at low loads but rises at higher loads. Thus dependency extraction can be

performed at low to medium loads and stopped, if the precision level of the graph is to be maintained. Table 2 shows that a higher polling rate[4] leads to increased management overheads. Table 3 suggests that higher load implies higher overhead of polling.

6 Conclusion

In this paper we propose a new method for discovering dependencies between system components using non-intrusive techniques. To demonstrate the usability of the method we performed extensive experimentation on an e-business setup that mimics the operation of a typical storefront system using the TPC-W benchmark. The algorithm for computing the dependencies is experimentally shown to perform well with accuracy being 100% both at low and high loads, and precision decreasing with increasing load. For instance the bookstore application the precision is around 98% for a load of 50 simultaneous customers.

The nature of the algorithm is such that in addition to discovering all the true dependencies, it also discovers false dependencies. To preclude usage of the false dependencies we proposed a sorting heuristic, which increases the probability of placing the true dependencies higher in the sorted list. As expected, our experiments show that the precision increases as load decreases. A decrease in load causes a decrease in concurrency in the system and hence decreases the chance of discovering a false dependency. Thus, false dependencies can be artificially minimized if sufficient time gap exists between two successive transactions. In order to present various views of the dependencies to a system administrator, the dependencies of a component can be sorted in a manner that also uses the execution time or the frequency of invocation, or any other attribute of the antecedent components. We intend to look into such issues in future. Last but not the least, we intend to explore how to relax the containment assumption and capture dependencies in an asynchronous transaction environment.

Acknowledgements

The authors like to thank Yuan Feng (IBM Watson Research Center), Sugata Ghosal (IBM India Research Lab.) and Parviz Kermani (IBM Watson Research Center) for critical feedback of the paper.

References

[1] J. Aman, C.K. Eilert, D. Emmes, P. Yocom, 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 (ARM)," OpenGroup Technical Standard C807, UK ISBN 1-85912-211-6 July 1998, http://www.opengroup.org/products/publications/catalog/c807.htm

[4] Management data from WAS is obtained through polling at regular intervals. See [11].

[3] S. Bagchi, G. Kar, 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] Brown, G. Kar, and A. Keller, "An Active Approach to Characterizing Dynamic Dependencies for Problem Determination in Distributed Environment," *International IFIP/IEEE Symposium on Integrated Network Management*, 2001.
[5] 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.
[6] 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 Communication*s, Vancouver, BC, Canada, 1999, pp. 1547–51.
[7] CIM: http://www.dmtf.org/standards/standard_cim.php
[8] Ensel, Christian, "New Approach for Automated Generation of Service Dependency Models," *Second Latin American Network Operation and Management Symposium, LANOMS,* 2001.
[9] J. A. Farrell and H. Kreger, "Web services management approaches," *IBM Systems Journal*, VOL 41, NO 2, 2002.
[10] Gruschke, "Integrated Event Management: Event Correlation using Dependency Graphs," Proceedings of 9th IFIP/IEEE International Workshop on Distributed Systems: Operations and Management (DSOM 98), October 1998.
[11] M. Gupta, A. Neogi, M. K. Agarwal, and G. Kar, "Discovering Dynamic Dependencies in Enterprise Environments for Problem Determination," *IBM Research Report*, RI03010, 2003.
[12] P. Hasselmeyer, "Managing Dynamic Service Dependencies," Proceedings of 12th International Workshop on Distributed Systems: Operations & Management (DSOM) 2001.
[13] J.L. Hellerstein and S. Ma, "Mining Event Data for Actionable Patterns," *The Computer Measurement Group*, 2000.
[14] Java 2 Platform, Enterprise Edition, http://java.sun.com/j2ee
[15] M. J. Katchabow et al., "Making Distributed Applications Manageable Through Instrumentation," *Journal of Systems and Software*, Vol. 45, 1999.
[16] S. Katker and M. Paterok, "Fault Isolation and Event Correlation for Integrated Fault Management," *Integrated Network Management V, Chapman and Hall*, May 1997.
[17] Keller and G. Kar, "Classification and Computation of Dependencies for Distributed Management," *5th IEEE Symposium on Computers and Communications (ISCC)*, July 2000.
[18] F. Kon and R.H. Campbell, "Dependence Management in Component-Based Distributed Systems," *IEEE Concurrency*, Vol. 8, No. 1, pp. 26-36, Jan-Mar 2000.
[19] M. Steinder and A.S. Sethi, "Multi-layer Fault Localization using Probabilistic Inference in Bipartite Dependency Graphs," Technical Report 2001-02, CIS Dept., Univ. of Delaware, Feb 2001.

[20] Thoenen, J. Riosa, and J. L. Hellerstein, "Event Relationship Networks: A Framework for Action Oriented Analysis for Event Management," *Proceedings of the IFIP/IEEE International Symposium on Integrated Network Management*, Seattle, WA, May 2001, IEEE, New York (2001), pp. 593-606.
[21] TPC-W Wisconsin website, http://www.ece.wisc.edu/~pharm/tpcw.shtml
[22] S. Yemini, S. Kliger et al., "High Speed and Robust Event Correlation," *IEEE Communications Magazin*e, vol. 34, no. (5), pp. 82–90, May 1996.