

Root Cause Isolation for Self Healing in J2EE Environments

Umesh Bellur, Amar Agrawal
umesh, amar@it.iitb.ac.in
School of Information Technology, IIT Bombay

Abstract—The increasing complexity of distributed enterprise systems has made the task of managing these systems difficult and time consuming. The only way to simplify the management process is to automate much of the work so that minimum human effort could be invested. This has led to research in self-managing or autonomic systems that are self-healing, self-configuring, self-protecting and self-managing. We believe that the starting point of any autonomic system is to understand the dependencies between various components of the system and use it to perform higher order management tasks. As a proof of concept, we are trying to build self-healing capabilities into distributed enterprise applications by modelling the failure dependencies within the system. In a complex distributed environment, failures tend to propagate from one part of the system to the other. Hence, the failure symptoms may be observed at a point far removed from the actual cause of these failures. Therefore, localizing observed failures to its root cause is an important pre-requisite to initiate micro-recovery procedures on a failed system. Under this project, we develop a methodology to obtain a failure model of the application and use it to perform root-cause analysis based on observed system failures.¹

I. INTRODUCTION & MOTIVATION

Modern distributed enterprise systems are complex enough to warrant the automation of management tasks. The lack of human expertise and the flood of monitoring data that these systems generate has led researchers to build self-managing capabilities in these systems. The resulting self-healing, self configuring, self-protecting and self-optimizing systems - also known as autonomic systems - try to adapt themselves to changes (failures, reconfiguration, addition/deletion of components, etc.) in a way that requires minimum human intervention. Self-healing systems are a class of autonomic systems that are able to actively recover from failures. A distributed application is composed of various software and hardware components. These components interact in complex ways to provide the required functionality to the user. Dependencies therefore naturally exist between these components, due to which a fault in one of them tends to propagate to other parts. This may also result in multiple symptoms being thrown by different other components in the system. The self-healing process involves the following steps:

- 1) Monitor the application for failure symptoms
- 2) Localize these symptoms to root-cause faults
- 3) Initiate appropriate recovery procedure

¹This project was sponsored in part by a IBM Faculty Award for Autonomic Computing in 2006 and a generous research grant in Autonomic Computing from the IT Research council of Intel Corp

Monitoring is largely infrastructure and application dependent. The system should be monitored for all such failure symptoms that can be used to diagnose the faults in the system. These include system/application exceptions, performance violations, user-visible errors, timeouts, etc. The monitoring is to be deployed on a live production system and hence must be as non-intrusive as possible. There is a tradeoff involved between the granularity of monitoring data available from the system and the intrusiveness of the monitoring mechanism used. Assuming that a good monitoring system is in place, the next step of the self-healing process is to process the failure data from the system in order to pinpoint a set of components that are the root-cause of these failures. This usually requires a failure model of the system to be built before hand, which is then used for the failure diagnosis process. This is a crucial step and is useful in directing the recovery efforts towards components that have a higher probability of failure, given the information about the observed failure symptoms. Accurate root-cause analysis would mean more effective and quicker recovery. Once the responsible failed components have been identified, suitable recovery procedures can be initiated to deal with the failures at hand. The possible recovery actions are system specific and may require rebooting the failed components for transient failures, replacing or switching to a backup component or just isolating the failed parts while reducing the available functionality. Under this project we look at the root-cause analysis problem, while building the required monitoring to demonstrate a proof-of-concept system. Listed below are some important requirements for the failure diagnosis

- Must be able to handle inconsistencies in the fault model as well as the system state.
- Must be able to handle multiple simultaneous faults with possibly overlapping fault symptoms.
- Performance-related faults must be appropriately modelled and diagnosed.
- Be event-driven so that inaccuracies due to incorrect time-window specifications specific to window-based techniques could be avoided.
- Have high accuracy and completeness with low computation complexity.

Several approaches to root-cause analysis have been proposed in literature. Most of these are expert systems that use past failure data to build a failure database. Observed failure symptoms

are mapped to known fault cases or against rules composed out of this database to identify current faulty components. Such approaches usually require huge amounts of failure data that cover the entire failure space and insufficient for unknown or multiple fault scenarios. There are other approaches which use correlation based analysis to find those components that are most correlated with failed requests. No failure data is assumed, but usually results in a much bigger faulty set which is not very useful. We believe that, by modelling the failure dependencies in the distributed system, we can do better diagnosis. The propagation of failure along the call stack can be captured using the topology of the application. The assumption here is that the failure dependencies mirror usage dependencies, i.e. faults propagates to a component only when it uses a failed component either directly or through some other mediating component. To handle the possibility of uncertainty, in terms of lost or spurious symptoms, we adopt a probabilistic methodology using Bayesian Belief Networks (BBNs). The diagnosis task is largely event driven where the stream of failure events generated are fed to the BBN model. The model outputs ranked set of components together with a probabilistic quantification of the belief in their failure given the failure symptoms so far.

II. RELATED WORK

Many fault diagnosis techniques have been proposed over the years. These methodologies can be classified on the basis of whether they use the dependency information for the fault localization process. Some systems such as expert systems[1], [2] and the newly proposed machine learning techniques do not directly use topology information for the diagnosis while others use dependency graphs and fault propagation models[3], [4] to direct the fault localization process. We present some of the important related works in the following sections.

A. Non Dependency-based

In this section we discuss some of the approaches that do not explicitly use topology information for the fault diagnosis.

1) *Expert systems*: Expert systems have been traditionally used for fault localization and diagnosis. An expert system tries to mimic a human expert when solving problems in a particular domain. Expert systems applied for fault localization differ with respect to the structure of knowledge they use. Approaches that rely solely on surface knowledge resulting from past experience are called rule-based systems while those that use deep understanding of the underlying system are called model-based systems.

- **Rule-based reasoning (RBR)**: The knowledge base is in the form of if-condition- then-action rules. The condition depend on various events in the system and the system state while the action refers to the steps to be taken in response to the condition. Rule-based reasoning is very intuitive and is good for small, non-changing and well understood systems. However, these systems have several drawbacks such as inability to learn from experience, inability to deal with unseen problems and difficulty in

updating the rule base due to convoluted, hard-coded rules[1].

- **Model-based reasoning (MBR)**: Model-based reasoning incorporates deep knowledge of the system through use of an explicit system model representing causal, statistical or mathematical information about the system. Some of these models use the dependency information such as causality graphs. Others such as mathematical models may or may not use any dependency information. Fault diagnosis is performed by comparing the normal system model with observed system behavior to find discrepancies. Mathematical models are generally hard to build for large systems and are also computationally expensive to use.
- **Case-based reasoning (CBR)**: Case-based reasoning systems store past cases as well as recovery actions used for them to come up with solutions to the current problem. During diagnosis, if the current problem matches an already known case exactly then, the solution is trivially available. Otherwise, new solution is derived using past cases(e.g. closest match) and their solutions. Case-based systems tend to be application specific and case matching and solution construction is hard. Case- Line system uses a CBR approach for fault diagnosis and recovery for British Airways.

2) *Data Mining approaches*: Various machine learning techniques can applied to the problem of failure diagnosis. The underlying motivation for using these techniques for root-cause analysis is to find system components that are highly correlated with observed failures in the system. Client requests are traced through the system (through proper instrumentation if required) to determine the components used as well as the success/failure of the request. This information is then processed using machine learning techniques under the assumption that the components that are highly correlated with failures must be the root cause of observed system failures.

- **Decision Trees**: Using the trace data as described above, we learn a decision tree to classify the request as failure or success. The paths in the tree that lead to failure-predicting nodes are then post-processed to extract the relevant components. Decision trees are easy to interpret and are quick to learn but are known to be unstable. Inorder to reduce the number of false positives, leaves containing less than a certain percentage of total failures are ignored. For further details look at [3].
- **Clustering**: From the trace data we have for each request the set of components it uses. This data is transposed to obtain for each component, the requests that uses that component. Here, $a_{ij} = 1$ represents request j uses the component i , while $a_{ij} = 0$ represents otherwise. A failure data point is also added and marked with all requests that are believed to be failed. A clustering algorithm is then used to cluster all the components together. The set of components clustered together with the failure point are those that are most correlated with request

failures and hence more likely to be the root cause. A limitation of this approach is that it cannot distinguish between components that are tightly coupled e.g. if two components are always used together the real root-cause cannot be isolated. Pinpoint [4] uses clustering for fault diagnosis.

It will be interesting to compare the results obtained by our approach with those obtained using data mining techniques to verify whether topology information can be used for better diagnosis.

B. Dependency-based

These approaches explicitly use the dependency information in the application for diagnosing failures. This dependency information may be either the topology information (method invocation, resource dependencies, etc.) or the fault propagation graphs that indicate how fault in one part of the application propagates to other parts. Various approaches using such dependency knowledge have been proposed in the literature. Dependency information can also be used in the recovery procedure for determining the sequence in which the recovery actions must be initiated.

1) *Minimum Set Cover*: This is a deterministic approach to failure diagnosis which considers only complete failures i.e. a component either fails completely or not. The problem then reduces to finding the optimal set of failed components that could explain the set of observed symptoms. The optimization measure could be the cardinality of the fault set or the cost of repairs or any other suitable criteria. The problem of minimum set cover is NP complete and there are approximate algorithms to compute the minimum set cover. Greedy approaches such as forward addition and backward eliminations can lead to suboptimal fault sets. The problem could be formulated in multiple ways. Every request could be traced through the application to determine its success/failure status and the components it uses (together known quark). Then the set of faulty components is the minimum set cover over the faulty quarks. Else, if a fault database is available with information on the observable symptoms for each fault expressed as a dependency graph, then starting with the observed symptom nodes (marked) we can find the set of fault nodes that cover all the initial marked nodes. Kompella [5] uses this technique for link fault localization in IP and optical networks.

2) *Codebook Approach*: The codebook approach uses a dependency graph with two kinds of nodes viz. problem nodes (representing known faults) and symptom nodes (observable events). Event dependencies are indicated by directed edges in the graph. Weight could also be assigned to these edges indicating the probability that a fault/event A causes event B. The initial dependency graph may be pruned to remove cyclic dependencies, indirect symptoms, etc. to obtain a digraph. This digraph is then encoded into a correlation matrix with faults at the columns and events at the rows. Probabilities corresponding to edges in the graph are put in the matrix. Thus each fault is represented as a vector of observable events in the system which is called the code for that fault. The

diagnosis problem is reduced to finding the set of faults whose symptoms or codes match closely the observed set of events. SMARTS Incharge [6] is one such system that uses codebook approach for fault diagnosis. The codebook approach requires that each fault be encoded in terms of the symptoms that are observable when the fault occurs in the system. This would require the knowledge of all expected failures in the system and a technique to inject these failures into the system in an isolated environment in order to observe the symptoms that are generated. The problem of fault code matching becomes harder when multiple simultaneous faults can occur in the system and the components can have partial failures. Lost or spurious symptoms can lead to false diagnosis which is related to the hamming distance between fault codes.

3) *Fault Propagation/Causality Graphs*: A causality graph is a directed acyclic graph whose nodes correspond to events and edges describe the cause-effect relationship between the events. A fault propagation graph (FPG) is a causality graph with faults and symptoms as nodes and edges represent how faults propagate throughout the system. These can be obtained by injecting suitable faults in the system and updating the FPG as fault propagates through the system as in [7]. A dependency graph essentially represents the logical dependence between the components in the system. The dependency graphs can be obtained by tagging each request to the system tracing its path through the system as in [8]. Performance-related dependencies could be obtained through active perturbations in the system and using statistical modelling to compute dependency strengths [9]. JAGR [10] uses fault propagation graphs for determining the set of components that the observed failures might have propagated to and restarts these components. Gruschke [11] uses a dependency graph to localize a set of managed objects that would explain a large number of observed events.

4) *Problem determination using active probing*: Probes are end-to-end test transactions which gather information about system components. Active probing allows probes to be selected and sent on-demand, in response to one's belief about the state of the system. As probe results are received, belief about the system state is updated using probabilistic inference. This process continues until the problem is diagnosed. Irina Rish [12] work on active probes uses a bipartite Bayesian belief network to connect probe outcomes with component states. Each probe can detect the state of some components which can be represented as a two dimensional matrix. The system assumes the existence of pre-determined probe sets that cover all the components. Unimodal (Failed Vs Not Failed) failure of components is assumed and the failure space increases exponentially when multiple simultaneous faults are considered. The bipartite graph does not take into account the possibility of a failure triggering other failures. The approach is not suitable for software applications where each component can fail in varied number of ways and simultaneous correlated failures are common. Also, failures can cause other failures which cannot be captured by a bipartite dependency graph. If, requests are treated as probes, then the approach reduces

to determining the failed components based on correlation with failed requests. If probes are exception events, then the methodology reduces to the codebook approach where each fault has a component code.

C. Our Approach

Instead of relying solely on failure/success status of requests for determining the root- cause of failures, we use exceptions generated by the application. Essentially, we will use an expert system or a failure model, but that is not based solely on known fault cases or hard-coded rules. We will use the application topology information to account for the possibility of fault propagation from one component to the other. Bayesian belief networks [13] (BBNs) are probabilistic directed graphs which enable us to model the failure dependencies among components. The dependency strengths can be expressed in terms of apriori state probabilities associated with each node in the BBN. In other words, given evidence about the states of some components, we can infer the probability of other dependent components too. Both, causal and inferential belief propagation is possible. The system state is uncertain most of the times due to the possibility of loss or spurious data. The problem of reducing the observed symptoms to root-causes becomes easier with BBNs, which work well even with uncertain data. Multiple simultaneous failures are handled naturally with BBNs since there is no issue of determining the size of the fault set. The BBNs provide a ranking of the failed components. If two components have failed, they will be higher up in the ranking order. BBNs also provide an event based diagnosis technique where the stream of fault symptoms is fed to the model and the state of the system is updated.

III. BBN GENERATION METHODOLOGY

In this section, we discuss the methodology of generating the BBN model and using it for failure diagnosis. The BBN model is based on the topology of the application. The term "Topology" represents:

- Physical, logical infrastructure and their configuration.
- The static and dynamic view of the application components. By static we refer to the interfaces supported by each component and its packaging details while by dynamic we refer to the deployment view of components relative to one another. This is necessary to create analytical or simulation models for the purposes of self configuration eventually.
- Dependencies that exist between the application components as well as those between application components and infrastructure (software, hardware and network).

The topology graph gives the usage dependencies between the application components as well as usage probabilities between them. This is supplemented with exception propagation analysis of the application code to generate the failure dependency graph. This graph is encoded into a BBN for easier diagnosis of observed failure symptoms. The rest of this section details these steps.

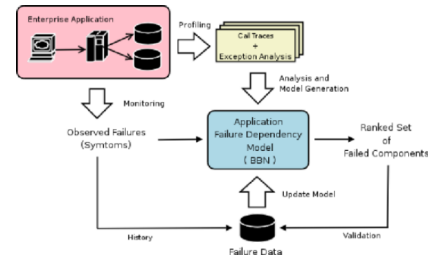


Fig. 1. Failure Diagnosis Tool Architecture.

A. AutoTopology : Generating Call Traces

The AutoTopology tool extracts usage dependencies between the software components of any J2EE application. It uses the JVMPi interface provided by all JVM implementations and the concept of Interceptors in J2EE application servers to generate call traces for the application. The topology for the application is generated out of the call traces by executing the use-cases for the application and combining the dependency information in these call traces. The tool generates the call traces through following steps:

- 1) Profiling and logging from within the JVM using JVMPi
- 2) Interceptors and logging at the container level.
- 3) Node analysis to establish traces using these logs at each node
- 4) Merging call traces across nodes to establish a single trace.

B. TopoGenerator : Generating the Topology Graph

The call traces generated by AutoTopology tool is then aggregated into a topology graph. The topology graph is directed graph with nodes representing the components in the application and direction of the edges indicating the usage dependency. For example, if a component A uses component B then the graph will have an edge (A,B) representing this dependency. Each edge is also annotated with a dependency strength that represents the probability of a component A using another component B in the application. Following cases in the original call trace need to be considered while transforming them into topology graphs:

- 1) Sequential path : A sequential call-edge is mapped directly to corresponding usage edge in the topology graph with the usage probability remaining the same as that before the call is made.
- 2) Alternate path : When the call trace has an AlternatePath node, the execution can take any of the available alternate paths and the probability of usage splits at this node. If no additional information regarding the probability distribution of taking these alternate paths is known, a uniform distribution is assumed. If the same call-edge occurs multiple times in a call trace, then the probability of making that call needs to be accumulated to generate a single value for the usage probability. Consider the following interesting cases:

Algorithm 1 GenerateTopology(callTree, usageProbability)

```
1: for each child  $C$  of CallTree do
2:   if  $C$  is an AlternatePath node then
3:      $usageProbability = usageProbability/numberOfAlternatePaths$ 
4:     for all Alternate paths  $P$  of  $C$  do
5:        $altUsageEdges[P] = GenerateTopology(P, UsageProbability)$ 
6:        $childUsageEdges[C] = AggregateAcrossAlternatePaths(altUsageEdges)$  {ADD}
7:     end for
8:   else
9:      $childUsageEdges[C] = GenerateTopology(C, UsageProbability)$ 
10:  end if
11:   $usageEdges = AggregateAcrossChildren(childUsageEdges)$  {UNION}
12:  return  $usageEdges$ 
13: end for
```

- a) Nested alternate paths : If an alternate split is nested in some other alternate split, then the call-edge will be taken if both the alternate splits select the favorable path. Hence, the total usage probability is a product of their individual probabilities.
- b) Union across child subtrees : If the call edge occurs multiple times (but not nested) in the call trace, then total usage probability is the union of the events that both call-edges will be taken. If they belong to two different paths of the same alternate split, then the union operation will reduce to a simple ADD since the intersection of the events will be zero.
- 3) Concurrent path : Concurrent paths are executed in parallel, but are reduced similar to the sequential path case.
- 4) Loop path : The loop path is treated as a single sequential path. Algorithm 1 generates topology graphs for individual use-cases. The AggregateAcrossAlternatePaths procedure aggregates the usage probability for common usage edges across child alternate paths through a ADD operation while AggregateAcrossChildren aggregates usage probabilities across child sequential paths using a UNION operation.

Algorithm 1 generates topology graphs for individual use-cases. The AggregateAcrossAlternatePaths procedure aggregates the usage probability for common usage edges across child alternate paths through a ADD operation while AggregateAcrossChildren aggregates usage probabilities across child sequential paths using a UNION operation. The topology of the application can be generated by computing a weighted addition across all the use-cases using the probability of executing the usecase as weights. The use-case probabilities can be obtained from historical data or in the simplistic case assumed to be uniform across all usecases.

C. Exception Analysis

The failure dependency strength is a function of usage dependencies between the various components and the failure propagation probabilities between components. The usage

probabilities are obtained from aggregating the usages across all call traces. Failure propagation probabilities can be obtained either through available failure data if available or using static analysis on the application code. We use JeX, an exception visualization tool, that helps to determine the dependencies between various exceptions that the application can throw and their associated probability values. The probability of an exception E_1 propagating from one component A to the other component in the form of exception E_2 can be estimated statically as:

$$P(E_1 \text{ triggers } E_2) = P(B \text{ uses } A) * P(E_1 \text{ triggers } E_2 | B \text{ uses } A) \quad (1)$$

In the context of JAVA components failure manifests as exception and can be used synonymously with failure of the component. The propagation of exceptions from one component to the other can be visualized using an exception visualization tool like JeX. JeX analyzes JAVA source code and class files to generate jexfiles that help in visualizing the exception flow through the system. The JeX file indicates what exceptions can be raised in each method of a class. The usage probability $P(B \text{ uses } A)$ comes from the topology graph while static analysis using JeX files gives the second probability in the above equation.

D. Generating the Bayesian Belief Network

We represent every exception a component can throw as a node in the BBN. We then connect those exceptions that have a propagation dependency between them. For example, consider a component B that uses another component A in the topology graph. Also, A can throw an exception E_1 that triggers an exception E_2 in component B . Thus, exception E_1 can propagate to component B and we add an edge from node $A :: E_1$ to node $B :: E_2$ in the BBN. The dependency strength for this edge represents the probability of the exception propagation which is determined as described in Section III-C. Under the *Noisy-OR* model, this dependency strength maps directly to $P(E_2 | E_1)$ in the CPT of node E_2 . The *Noisy-OR* model also requires a leak probability which

is the probability of an exception occurring given that all its causes are absent. We choose a random low value for this leak probability. The BBN also contains some ‘decision’ nodes which are basically known fault cases that can be mapped to one or more exceptions in the BBN. For instance, in a J2EE application scenario, a *FinderException* thrown by an entity bean is indicative that there is more than one entity with the same primary key. For this case, we can add a ‘Duplicate Entity Entries’ fault node in the BBN with an edge to the *FinderException* node. The prior probability associated with all such nodes can only be determined through a failure database. We also add nodes representing various use-cases in the application. Information regarding failure of a use-case can be fed by instantiating the node with its state set to *Failed*. In case of loss of symptoms down the call stack, this provides some information regarding the possible nodes responsible for failure.

E. Monitoring : JVMDI Agent

Once the BBN model for the application is ready, we can monitor the application for failure symptoms. We make an assumption here that various software and hardware faults in the system manifest themselves as exceptions in the application being monitored. For example, a crash of a node on which an application server is loaded, might cause *RemoteException* in those EJBs that use components deployed on that server. In order to extract all such exception events being generated by the application, we require a monitoring module that can catch exception events. These failure events are fed to the BBN as evidence. We have written a JVMDI agent for Java applications, that can register for exception events being generated in the JVM. These events are filtered as per requirements to generate a stream of exceptions that are logged. We induce faults into the application and the system on which it is deployed to generate test cases for evaluation of our approach.

F. Inferencing on Failure Evidence

The failure events generated by the monitoring unit are fed to the bayesian model as evidence. Inferencing algorithms for the Bayesian belief network are used to propagate this evidence to other nodes. The belief in the state of all nodes is updated so that each node has some posterior probability of failure given the failure evidence. A ranking of the components based on this posterior probability is produced.

In a test environment, the output of the model is evaluated against known fault cases for correctness and completeness of the diagnosis. In a production scenario, the ranking helps in locating the root-cause of failures and directing recovery efforts towards components that are believed to be the most faulty.

In the next section, we present a case-study of a sample J2EE application - ECPeef. We apply the process described in this chapter to obtain a BBN for ECPeef. We then evaluate the diagnosis capabilities of the BBN through a few test scenarios

by inducing faults in the system and feeding the observed exceptions as evidence into the BBN.

IV. CASE STUDY: THE EC PERF SUITE

ECPeef [14] is a standard benchmark application for J2EE application servers. We use it here as a case study to demonstrate our approach to failure diagnosis. An ECPeef implementation is deployed on two JBOSS application servers, the web tier on one machine and the business tier on the other. A MySQL server is used as backend database system and holds the business data. We use GeNIe [15], a Bayesian belief network tool for modelling and inferencing.

We exercise the following usecases to generate the application topology:

- *New Order*: Generates a new order for a customer. The items to be ordered are added to a basket and the order is made.
- *New Order using Session Beans*: Stateful session beans are used to make a new order for a customer.
- *Change Order*: The customer can change an order by adding or deleting items.
- *Cancel Order*: An order can be cancelled using its unique order number.
- *Order Status*: Check the details of an order using its unique order number.
- *Customer Status*: List details of all orders made by a customer.

Call traces for all the use-cases are then aggregated into a topology graph for the ECPeef [Figure 2].

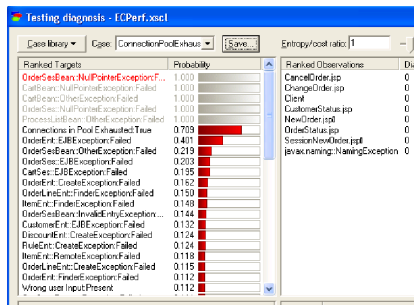


Fig. 5. Failure Scenario 3 : Connections in the managed DB connection pool are exhausted.

C. Multiple Simultaneous Faults: Two EJBs - ItemEnt and CartSes - are Undeployed

The diagnosis output shows fault in OrderLineEnt with highest probability while CartSes is very low down in the fault ranking (Figure 6). This is due to the fact that the CartSes bean is used with very low probability, hence evidence from CartBean has low priority over other evidence.

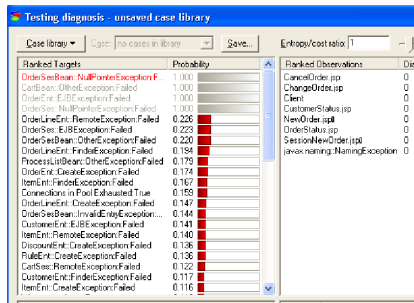


Fig. 6. Failure Scenario 4 : Multiple Faults: OrderLine and CartSes undeployed.

VI. SUMMARY

Failures tend to propagate from one part of the system to another due to the dependencies in the system components. Hence, root-cause analysis is a crucial step for self-healing systems where the observed failure symptoms are localized to a root fault set.

Our approach to the problem of root-cause analysis uses the application topology to determine the failure dependencies within the system. These dependencies are quantified in terms of failure propagation probabilities from one component to another. We use Bayesian belief networks to encode the failure model and use it to process the stream of failure events obtained by monitoring the application. Given evidence about states of some of the nodes in the BBN, we update our belief in the states of other nodes.

The process of obtaining the BBN from call-traces of the application and static exception analysis of the application code has been outlined. We demonstrated a proof-of-concept system for a sample J2EE application, ECPperf. We do not require a huge failure database to construct the failure model.

Instead, as more failure data becomes available, the model parameters can be adjusted.

VII. FUTURE WORK

Through this project we have explored the possibility of using application dependency information for failure diagnosis. A map of future research directions is presented below:

• Implementation Extensions

- **Automation of BBN generation:** The BBN has been constructed manually using GeNIe, although some parts of the process, like topology graph extraction, have been automated. The process of generating the BBN and entering evidence into the model could be automated.
- **Failure classification:** The test case set could be expanded to consider other classes of failure like performance faults, SLA violations, timing errors etc. A module to automatically inject faults in the system and analyze the output of the BBN with known fault cases would be helpful.
- **Monitoring:** A monitoring module that could generate failure events other than exceptions would enable us to evaluate the diagnosis methodology for other classes of failures.

• Conceptual Extensions:

- **System dependencies:** The application dependencies are well understood and can be determined using static and dynamic analysis techniques. A way of extracting system dependencies and their interaction with the application needs to be explored.
- **Request Context:** We process failure events independent of the requests that cause them. There is some loss of information here and the diagnosis would be more accurate if we could associate a request context with each event. This would mean additional overhead for monitoring the system.

REFERENCES

- [1] L.M. Lewis. A Case-Based Reasoning Approach to the Resolution of Faults in Communication Networks. *Proceedings of the IFIP TC6/WG6. 6 Third International Symposium on Integrated Network Management with participation of the IEEE Communications Society CNOM and with support from the Institute for Educational Services*, pages 671–682, 1993.
- [2] RV Magaldi. CBR for troubleshooting aircraft on the flightline. *Case Based Reasoning: Prospects for Applications, IEE Colloquium on*, page 6, 1994.
- [3] M. Chen, AX Zheng, J. Lloyd, MI Jordan, and E. Brewer. Failure diagnosis using decision trees. *Autonomic Computing, 2004. Proceedings. International Conference on*, pages 36–43, 2004.
- [4] Mike Y. Chen, Emre Kiciman, Eugene Fratkin, Armando Fox, and Eric Brewer. Pinpoint: Problem Determination in Large, Dynamic Internet Services. In *DSN '02: Proceedings of the 2002 International Conference on Dependable Systems and Networks*, pages 595–604, Washington, DC, USA, 2002. IEEE Computer Society.
- [5] Ramana Rao Kompellay, Jennifer Yates, Albert Greenberg, and Alex C. Snoeren. IP Fault Localization Via Risk Modeling. In *NSDI '05, Proceedings of the 2nd Symposium on Networked Systems Design and Implementation*. USENIX, May 2005.

- [6] S. Klinger, S. Yemini, Y. Yemini, D. Ohsie, and S. Stolfo. A coding approach to event correlation. *Proceedings of the fourth international symposium on Integrated network management IV table of contents*, pages 266–277, 1995.
- [7] George Candea, Mauricio Delgado, Michael Chen, and Armando Fox. Automatic Failure-Path Inference: A Generic Introspection Technique for Internet Applications. In *WIAPP '03: Proceedings of the The Third IEEE Workshop on Internet Applications*, page 132, Washington, DC, USA, 2003. IEEE Computer Society.
- [8] Sudhir Reddy. Dynamic Topology Extraction for Component Based Enterprise Application Environments. Master's thesis, Kanwal Rekhi School of Information Technology, IIT Bombay, 2005.
- [9] A. Brown, G. Kar, and A. Keller. An active approach to characterizing dynamic dependencies for problem determination in a distributed environment. *Integrated Network Management Proceedings, 2001 IEEE/IFIP International Symposium on*, pages 377–390, 2001.
- [10] George Candea, Pedram Keyani, Emre Kiciman, Steve Zhang, and Armando Fox. JAGR: an autonomous self-recovering application server. *Autonomic Computing Workshop, 2003*, pages 168–177, 2003.
- [11] Boris Gruschke. Integrated Event Management : Event correlation using dependency graphs. In *Proceedings of the 9th International Workshop on Distributed Systems Operation and Management*. IEEE, 1998.
- [12] Irina Rish, Mark Brodie, Natalia Odintsova, Sheng Ma, and Genady Grabarnik. Real-time Problem Determination in Distributed Systems using Active Probing. In *Proceedings of NOMS-2004, Seoul, Korea*, April 2004.
- [13] About Bayesian Belief Networks, for BNet.Builder Version 1.0 - Charles River Analytics, Inc. 2004.
- [14] S. Microsystems. ECperf Specification, 2001.
- [15] GeNIe, The Bayesian Net Tool.