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 needs to be invested. This has lead to research in autonomic systems that are self-healing, self-configuring, self-protecting and self-securing. 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 modeling 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 prerequisite to initiating micro-recovery procedures on a failed system. In this paper, we suggest a methodology to obtain a failure model of the application and use it to perform root-cause analysis based on observed system failures.  

1. Introduction

The lack of human expertise and the flood of monitoring data that complex modern enterprise 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 can actively recover from failures. The self-healing process involves the following steps:

1. Monitoring the application for failure symptoms.
2. Localizing these symptoms to root-cause faults.
3. Initiating appropriate recovery procedures.

The various software and hardware components, that make up an application, interact in complex ways to provide required functionality to the user. Dependencies naturally exist between these components, due to which a fault in one of them tends to propagate to other parts and multiple symptoms being thrown throughout the system. Localizing these symptoms to the root-cause of the failure is crucial prerequisite to initiating any recovery procedures. Accurate root-cause analysis would mean quicker and more effective recovery. 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.

Several approaches to root-cause analysis have been proposed in literature. Expert systems[6, 7] that use past failure data to build a failure database have been traditionally used. Observed failure symptoms are mapped to known fault cases to identify current faulty components. Such approaches require huge amounts of failure data that cover the entire failure space and are usually insufficient for unknown or multiple fault scenarios. There are other approaches which use correlation-based analysis[3, 4] to find components that are most correlated with failed requests. No failure data is assumed, but results in a much bigger faulty set which is not very useful. For a detailed survey of related work, refer [2].

We believe that, by modeling 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 a ranked set of components based on a probabilistic quantification of the belief in their failure given the failure symptoms so far.

2 Our Approach

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.
- 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.

2.1 Generating Call Traces

The AutoTopology[9] 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.

2.2 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. Edge (A→B) is also annotated with a dependency strength that represents the probability of component A using component B in the application. Following call scenarios 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 equal to the probability of reaching the caller node.

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.

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.

The topology of the application can be generated by computing a weighted addition across all the use-cases using the probability of executing the use-case as weights. The use-case probabilities can be obtained from historical data or in the simplistic case assumed to be uniform across all use-cases.

2.3 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 or available or using static analysis on the application code. We use JeX[10], an exception visualization tool, that helps to determine the dependencies between various exceptions that the application can throw and their associated probability values. In the context of JAVA components failure manifests as exception and we use it synonymously with failure of the component.

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:

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

The usage probability $P(B \text{ uses } A)$ is derived from the topology graph while static analysis using $\text{JeX}$ files provides exception propagation probability $P(E_1 \text{ triggers } E_2|B \text{ uses } A)$ in the above equation.

### 2.4 Generating the BBN Model

We represent every exception that 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 2.3. Under the Noisy-OR[5] model, this dependency strength maps directly to $P(E_2|E_1)$ in the CPT(Conditional Probability Table) 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.

### 2.5 Symptom Monitoring

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.

We have written a JVMDI(Java Virtual Machine Debugging Interface) agent for Java applications, that can register for exception events being generated in the JVM. These events are filtered using class and/or method based filters to generate a stream of exceptions which are treated as failure symptoms in context of failure diagnosis.

### 2.6 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.

### 3 Test Setup and Experimental Results

ECPerf[8] is a standard benchmark application for J2EE application servers. We present preliminary test results in root-cause analysis using our approach for the ECPerf application. The test setup includes an ECPerf implementation 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 [1], a Bayesian belief network tool for modeling and inferencing. We generate a failure BBN model for the ECPerf application using the methodology described above. We induce faults in the application and log the exceptions generated. These are then fed to the BBN and
the predicted failure ranking is evaluated for correctness and completeness.

Table 1 lists a few test cases we used for evaluating the ECPef failure model. The failure probabilities predicted by the model for each of the components in the ECPef application given the observed failure symptoms is summarized for both single and multiple fault test cases. The component with the highest probability of failure has been marked in **bold**, while components that threw symptoms are underlined.

<table>
<thead>
<tr>
<th></th>
<th>Single Fault</th>
<th>Multiple Faults</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>ItemEnt Bean Undeployed</td>
<td>Connections in DB Pool Exhausted</td>
</tr>
<tr>
<td>Database</td>
<td>0.081</td>
<td>0.074</td>
</tr>
<tr>
<td>ConnectionPool</td>
<td>0.291</td>
<td><strong>0.709</strong></td>
</tr>
<tr>
<td>CustomerEnt</td>
<td>0.137</td>
<td>0.141</td>
</tr>
<tr>
<td>ItemEnt</td>
<td><strong>0.326</strong></td>
<td>0.232</td>
</tr>
<tr>
<td>OrderEnt</td>
<td>0.172</td>
<td>0.419</td>
</tr>
<tr>
<td>OrderLineEnt</td>
<td>0.239</td>
<td>0.149</td>
</tr>
<tr>
<td>DiscountEnt</td>
<td>0.134</td>
<td>0.128</td>
</tr>
<tr>
<td>RuleEnt</td>
<td>0.134</td>
<td>0.128</td>
</tr>
<tr>
<td>OrderSes</td>
<td>0.214</td>
<td>0.411</td>
</tr>
<tr>
<td>CartSes</td>
<td>0.115</td>
<td>0.108</td>
</tr>
<tr>
<td>ProcessListBean</td>
<td>NA</td>
<td>NA</td>
</tr>
<tr>
<td>CartBean</td>
<td>0.091</td>
<td>0.059</td>
</tr>
<tr>
<td>OrderSesBean</td>
<td>0.217</td>
<td>0.144</td>
</tr>
</tbody>
</table>

Table 1. Test Cases for ECPef Application

4 Conclusion

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.

We use 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 infer the failure states of system components using the stream of failure symptoms obtained by monitoring the application as evidence. We do not require a huge failure database to construct the failure model. Instead, as more failure data becomes available, it can be fed back to improve the apriori probabilities in the BBN model. Diagnosing multiple simultaneous faults is possible, as we evaluate the state of the entire system based on observed system behavior. The system can be extended naturally to include hardware failures as well as performance and QoS failures. For further details, refer [2].

References