Using Components for Architecture-Based Management

These slides use material from “Using components for architecture-based management: the self-repair case”, Sicard, Sylvain and Boyer, Fabienne and De Palma, Noel, ICSE '08: Proceedings of the 30th international conference on Software engineering.
Overview

- Introduction
- Architecture-Based Management
- Wrapping Legacy Elements
- Using Components for Management Services
- Experimental Results
- Conclusion
Introduction

- Architectural models as guidelines for management functions
  - Application deployment
  - Platform configuration
  - Reacting to node failure
- Use component-based technologies to build self-repairing management service
  - Existing technologies: JMX, SMARTFROG, JADE, RAINBOW, EGA
Introduction

- Component model exposes certain properties of the components
- Which properties determine the capabilities of a self-repair service built on that model
  - Which properties?
  - How do existing approaches satisfy these properties?
Introduction

- Consider the case of a J2EE application server
  - Distributed system
  - Architecture has multiple layers which must interact and combine legacy technologies
  - Self-repair for J2EE is still an open issue
Architecture-Based Management

- We consider the application level and the environment level
- Both levels may contain hardware or software elements
  - Management state of elements – set of properties exposed to the management service
  - Critical properties: local configuration, life cycle state, relationships between elements
- Example: Apache server
  - configuration specified in httpd.conf
  - started/stopped
  - connection to Tomcat server (if it exists)
J2EE Application Server

• Four layers
  – Web tier (Apache)
  – Servlet tier (Tomcat or other)
  – Enterprise JavaBean tier
    • Business logic
  – Database tier
J2EE Application Server

- Want to provide scalability and availability
  - Replication across clusters
  - Deal with machine failures
    - Detection
    - Restart failed replica on another node
    - Update connections
  - The management system must be aware of and able to alter the architecture of the system
Components

- Components play two major roles
  - Wrap legacy elements
    - Legacy elements do not expose properties in a uniform way
    - Wrap with coherent interface that the management service can make use of
  - Build the management system
    - The management system must also be able to repair itself
    - Constructing the management system itself from components is the first step
Components

• We abstract to the component model
  – Legacy elements become components
  – Elements comprised of smaller elements
    • Composition links replaced by a containment relationship on sub-components

• Requirements for abstraction
  – Attributes
  – Life cycle state
  – Interfaces
  – Bindings
  – Containment
Components

- Additional requirements for management service
  - Introspection
    - A way to dynamically discover the state of the five main requirements
  - Reconfiguration
    - Expose a way to modify the meta-data
    - Allows indirect manipulation of the legacy layer, provided that the management system reports these actions
Component-Based Repair Service

- Detect occurrences of well-identified failures
- Restore system to an active state
- More generally, maintain availability according to specified policy
- Repair service must repair itself as well as the applications it oversees
  - Self-healing behavior
Basic Repair Service

- Identify failed elements
- Get management state
- Replace failed elements with new elements that have the same management state
- Component status must be available after failure
  - Use a checkpoint mechanism to watch component status
Augmenting with Self-Healing

- Improve the reliability of management layer and the components of the Checkpoint layer
  - Simple redundancy of repair components is insufficient for autonomic operation
    - Every time a failure occurs, human operator must intervene and re-establish the number of redundant copies of the components
- Repair service must be able to re-establish this on its own, aside from simply replacing its own components
Augmenting with Self-Healing

- Components of the repair service must be under control of the repair service
  - Management layer should delimit “repair” area from “application” area
- “Replication should be provided as an orthogonal aspect allowing a component to be tagged with a “replicated” capability without having to program it specifically.”
Experiments

- JADE architecture-based management system
  - FRACTAL reflective, Java-based component model
  - FRACTAL components have two layers, membrane and content
    - Membrane defines abstractions, exposes the properties discussed previously
- Components enhanced to support replicated and checkpointed tags.
### JADE vs. ad-hoc

<table>
<thead>
<tr>
<th>Generic code</th>
<th># Java lines</th>
<th># ADL lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>Deployment Service</td>
<td>3505</td>
<td>1690</td>
</tr>
<tr>
<td>Checkpoint layer</td>
<td>6630</td>
<td>–</td>
</tr>
<tr>
<td>Replication layer</td>
<td>4567</td>
<td>832</td>
</tr>
<tr>
<td>Self-Repair service</td>
<td>4750</td>
<td>430</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>19452</strong></td>
<td><strong>2952</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Specific code</th>
<th># Java lines</th>
<th># ADL lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>J2EE</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Rubis app. - Web</td>
<td>150</td>
<td>11</td>
</tr>
<tr>
<td>Rubis app. - Servlets</td>
<td>150</td>
<td>11</td>
</tr>
<tr>
<td>Rubis app. - Database</td>
<td>150</td>
<td>11</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>450</strong></td>
<td><strong>33</strong></td>
</tr>
<tr>
<td>JMS</td>
<td></td>
<td></td>
</tr>
<tr>
<td>Apache Web server</td>
<td>800</td>
<td>16</td>
</tr>
<tr>
<td>Tomcat Servlet container</td>
<td>550</td>
<td>12</td>
</tr>
<tr>
<td>MySQL SGBD</td>
<td>760</td>
<td>40</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>2110</strong></td>
<td><strong>68</strong></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>JMS</th>
<th># Java lines</th>
<th># ADL lines</th>
</tr>
</thead>
<tbody>
<tr>
<td>JORAM server</td>
<td>368</td>
<td>51</td>
</tr>
<tr>
<td>JNDI</td>
<td>134</td>
<td>12</td>
</tr>
<tr>
<td>JMS Queue</td>
<td>253</td>
<td>16</td>
</tr>
<tr>
<td>JMS topic</td>
<td>297</td>
<td>16</td>
</tr>
<tr>
<td><strong>Total</strong></td>
<td><strong>1052</strong></td>
<td><strong>95</strong></td>
</tr>
</tbody>
</table>

*Table 1: Generic code vs. specific code*
Performance overhead of JADE

- Slight memory overhead
- Almost no response time or throughput overhead
- Significant availability improvement with JADE
- Human operator in scenarios w/o JADE was an expert who was waiting specifically for failures to occur
- In real-world scenarios, operator error accounts for most of the MTTR—up to 9 hours due to undetected failure or improper repair.

<table>
<thead>
<tr>
<th></th>
<th>Human admin.</th>
<th>JADE admin.</th>
</tr>
</thead>
<tbody>
<tr>
<td>Throughput</td>
<td>12 req./s</td>
<td>12 req./s</td>
</tr>
<tr>
<td>Resp. time</td>
<td>87 ms</td>
<td>89 ms</td>
</tr>
<tr>
<td>Mem. usage</td>
<td>17.5 %</td>
<td>20.1 %</td>
</tr>
</tbody>
</table>

Figure 6: Throughput, response time and memory usage of RUBiS with and without Jade

- nb. failed req. 1900
- MTTR 43 s
- Availability 0.96

- nb. failed req. 18700
- MTTR 464 s
- Availability 0.72

(a) With JADE  (b) Without JADE
Conclusions

- The set of properties laid out in the paper for self-healing are strong—they produce reliable self-healing behavior
- These properties are seldom satisfied by existing methods