Using Components for Architecture-Based Management
The Self-Repair case

Sylvain Sicard Université Joseph Fourier, Grenoble, France,
Fabienne Boyer Université Joseph Fourier, Grenoble, France,
Noel De Palma Institut National Polytechnique de Grenoble, Grenoble, France

ICSE '08 PROCEEDINGS OF THE 30TH INTERNATIONAL CONFERENCE ON SOFTWARE ENGINEERING
PAGES 101-110
ACM NEW YORK, NY, USA ©2008
TABLE OF CONTENTS ISBN: 978-1-60558-079-1 DOI>10.1145/1368088.1368103

Synopsis by:
Stephen Roberts, GMU CS 895, Spring 2013

Paper Intent

- Clustered J2EE application servers constitute an important and large segment of distributed computing
- The architectural complexity with several interacting tiers is rapidly surpassing the ability of humans to manage
- Require autonomic management services to increase or improve upon reliability and to rapidly adapt to change;
  - *specifically: an autonomic repair management service with self-healing behavior*
- Repair management *most depends upon* the properties provided by the component model
- Paper shares author's previous experiences with building an autonomic repair management service to self-healing behavior for J2EE and JMS middleware

Objectives: Architecture-Based Management System

- Architecture based management suggests managing two layers
  - Both Application & Environment
  - Each made of HW and/or SW elements:
  - Both layers exhibit "*management state*" which expose attributes to the control of the management system. *Essential aspects exposed*:
    - Local configuration settings
      - I.e. HTTPD server httpd.conf file settings
    - Life-cycle state
      - Started/stopped
    - Relationships between managed elements
      - Host Port/IP of connected Tomcat servlet server
- Global management state of a distributed application is an aggregation of the management states of its elements (both application and environment). Describes the current application architecture and configuration settings.
- Therefore:
- -> 1. Must be able to observe the management state of the application as an "open box"
- -> 2. Must be able to manipulate the management state
  - Add/delete elements
  - Modify aspects to include relationships

J2EE Example

- Need: J2EE specification aims to dynamically produce web pages to respond to client requests
- Must be scalable and highly available
- Composed of four tiers which can be dynamically distributed/configured -->
- Typical management scenario deals with failure of machines
  - Detecting failure
  - Automatically restarting failed element on another physical node
  - Updating element state for new node and connections
- Management scenario requires:
  - Knowledge of application server software architecture
  - Ability to manipulate the app server SW architecture

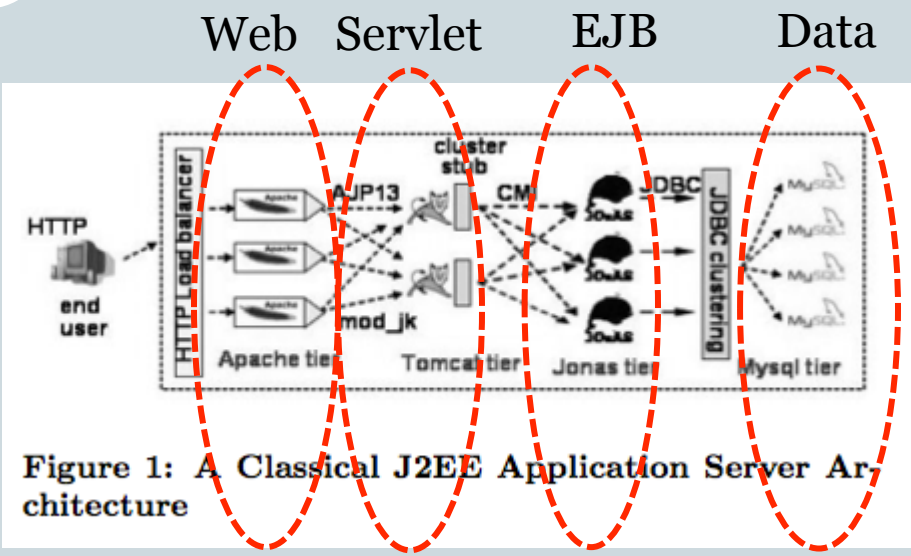


Figure 1: A Classical J2EE Application Server Architecture

- WEB and Servlet: execute app presentation tier (static and dynamically referenced pages)
- EJB: Business logic
- Data: Data storage/retrieval/manipulation

Architecture-based management component usage

- Two ways:
- Legacy element *wrappers* –
  - Used to realize the management interface of legacy elements and incorporate into a "*uniform management interface*"
    - Enables operations to observe and manage the legacy element
    - Designed in a standard fashion to simplify management service coding/design
- Building management services –
  - Benefit that management service applied directly to management service
    - Self-sizing service under control of repair management service used to automatically repair itself in case of failure

Component Legacy Wrapper Objectives

- Adhere to design principle - realizing management state of a legacy application into the component model's abstraction
- Legacy element *wrappers* –
  - Used to realize the management interface of legacy elements and incorporate into a "*uniform management interface*"
    - Enables operations to observe and manage the legacy element
      - Legacy element realized into a component and a composition link is realized into a containment relation between components
    - Designed in a standard fashion to simplify management service coding/design
      - Manipulation of legacy layer must go through management layer

Architecture of Legacy Wrapper Model

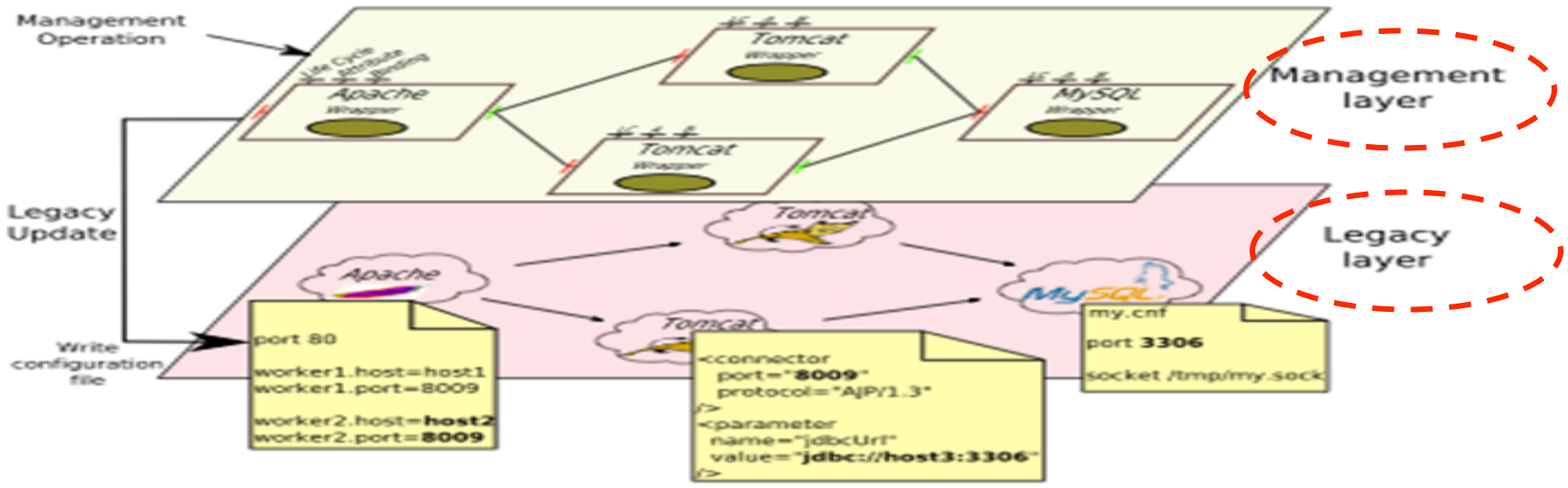


Figure 2: Basic Architecture of a Management System

- Main point: Two layers - new management layer which encapsulates legacy layer
  - current management state of legacy tier housed within management layer
  - any manipulation of legacy layer must go through management layer

Component Legacy Wrapper Abstractions

- Five attributes are sufficient to realize the management state of a legacy element
  - Attributes: enable the wrapper to observe/modify configurable properties (i.e. web servers configurable properties)
  - Life-cycle state: enable the wrapper to observe/modify legacy application state (i.e. start/ stop/new/delete)
  - Interfaces: enable the wrapper to observe/modify the functional dependencies to other managed elements (i.e. apache HTTPd web server and Tomcat servlet are to be executed as a unit)
  - Bindings: enable the wrapper to observe/modify the bindings between managed elements that have functional dependencies (i.e. creating and closing connections to other managed elements)
  - Containment: enable the wrapper to observe/modify the architecture composition links between legacy elements (i.e. Wrapper to a specific legacy element)

Component Legacy Wrapper

- Two-required classical meta-operations with the five abstractions
  - Introspection:
    - Components provide methods to dynamically discover their meta-data (current state of the 5 abstractions) -> observability
  - Reconfiguration:
    - Components provide methods to dynamically modify meta-data (add/ remove a binding) -> allows modifying the management state of legacy layer
- Typically requires operations to be *specializable* and a locking mechanism to prevent concurrency issues

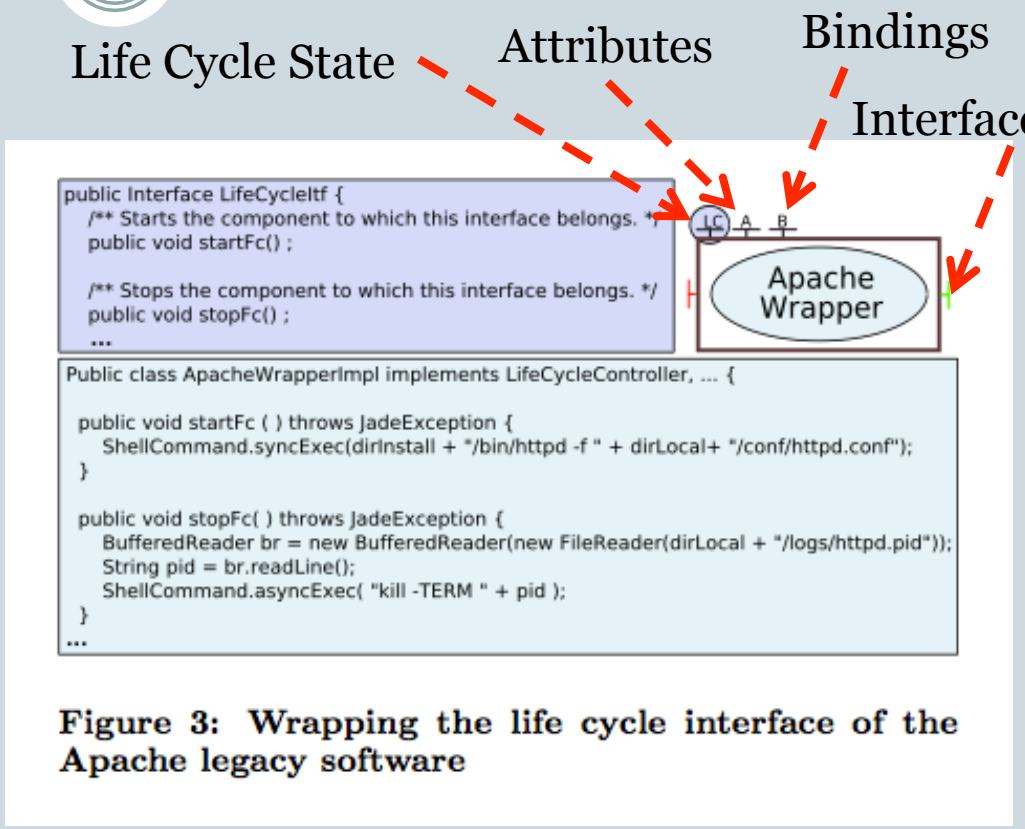


Figure 3: Wrapping the life cycle interface of the Apache legacy software

Architecture of Legacy Wrapper Model

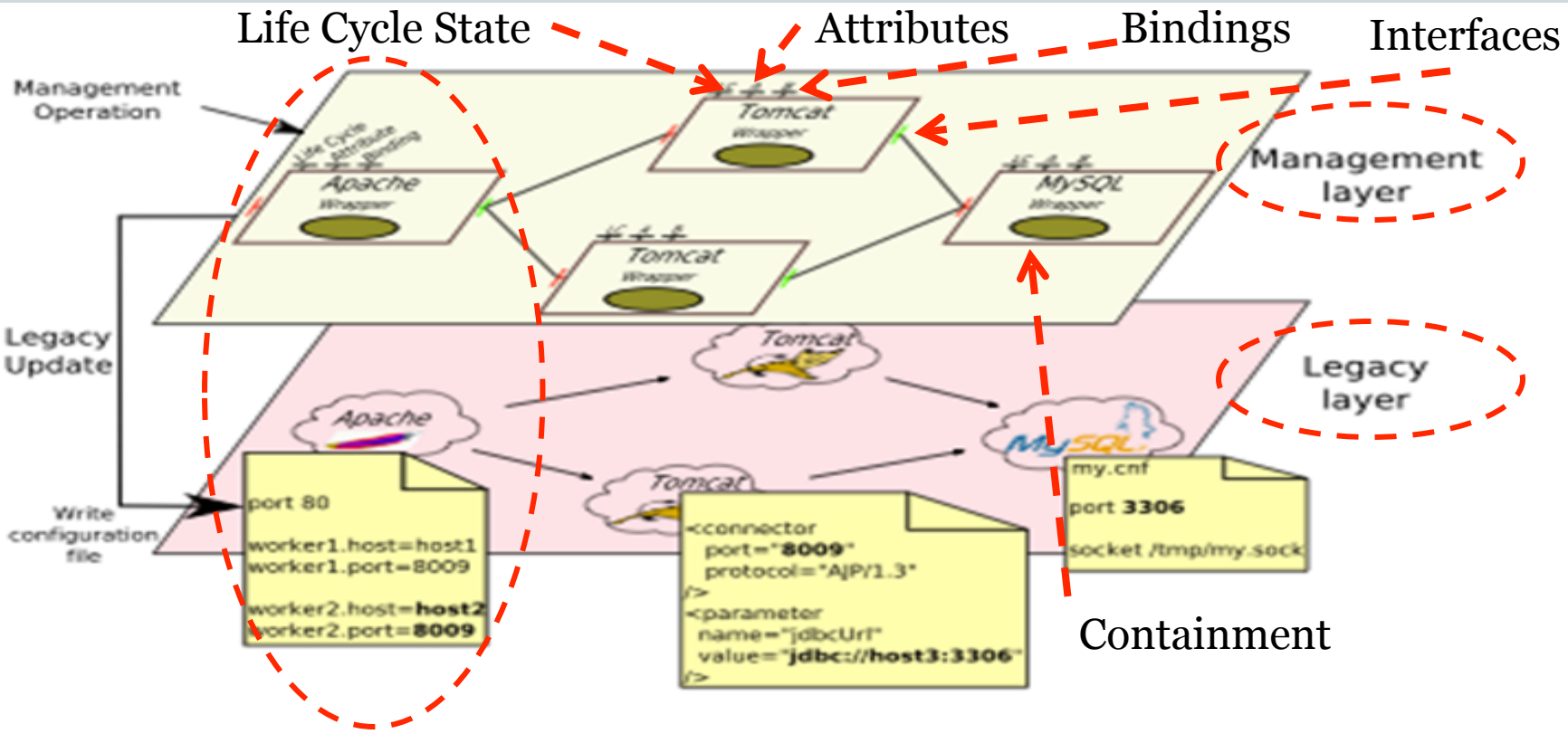


Figure 2: Basic Architecture of a Management System

Case Study: Autonomic Repair Service

- One of the most complex management services
- Goals:
  - Detect a well-identified type of failure
  - Restore a managed application to active state
  - To meet a stated level of availability
- Challenge:
  - Must be able to not only repair application but do so without human intervention
- Will next introduce requirements for:
  - A basic repair service
  - Then add self-healing capabilities

Autonomic Repair Service: Basic

- Core process steps executed after failure detection:
  - *Analysis* – identify failed element(s) and determine management state
    - Wrapper must be available to determine state of legacy element even after failure
    - Must utilize a checkpointing mechanism to provide up-to-date views of state
    - Introduces checkpointing layer
      - Made of components to track management and checkpoint layer
  - *Substitution* – substitute failed elements with newly initiated elements & modified with the same state
    - Requires reconfiguration step – bindings and containments
    - Bindings closed; new ones established to new components
    - If failed element was a sub-component of other components – containment relationship needs to be updated

Autonomic Repair Service: Basic

- Basic requirements introduce – *Checkpoint Layer*
- A meta-operation on a wrapper component (such as bind () on an apache wrapper) causes:
  - Checkpoint layer update: perform the same invocation in the checkpoint layer
  - Legacy layer update: invoke the specific management interface of the Apache legacy component

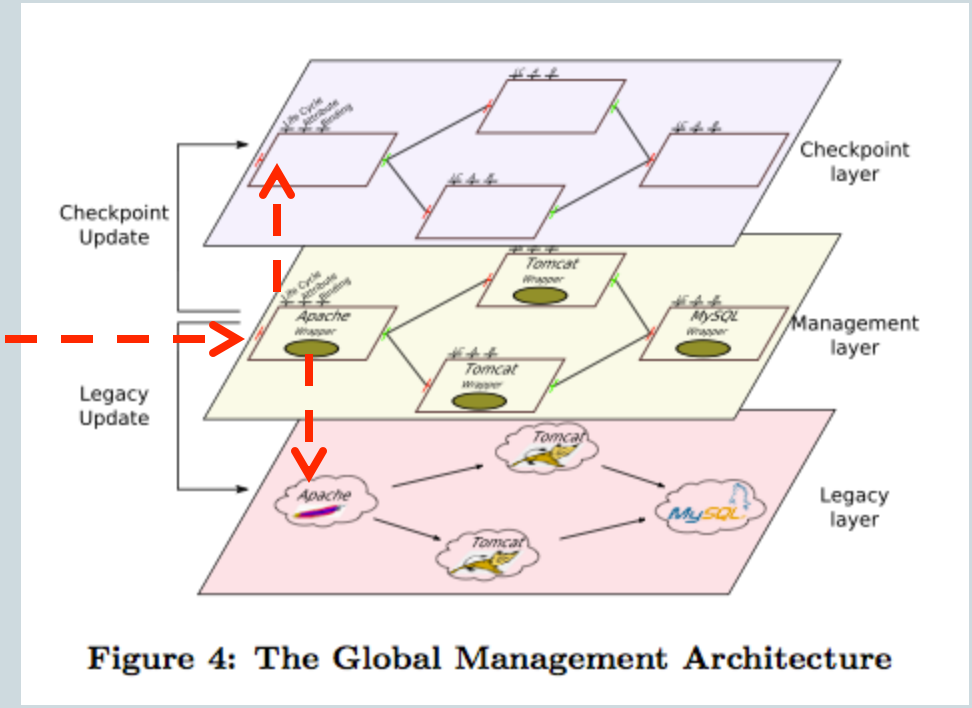


Figure 4: The Global Management Architecture

Autonomic Repair Service: Self-Healing

- Self-healing requires:
  - Add reliability to basic requirements just discussed
  - Add reliability to critical data accessed by repair service; (components of the Checkpoint layer)
- Generally: Reliability achieved via redundancy of both the core processes and checkpoint layer
- Self-healing still not achieved until:
  - Issue is with cardinality of replicas and removing need for human intervention to restore replica cardinality
  - Achieved using same algorithms and mechanisms used for repairing the legacy elements
    - Replicas of both core process of the repair service and the checkpoint must be under the control of the repair service i.e. has a representation in the management layer
    - Replication should be provided independently allowing a component to be tagged with a "replicated" capability without having to program it

Autonomic Repair Service: Self-Healing

- Components implementing the repair service as well as Checkpoint components are "replicated" components
- Replicas are represented/ referenced in the Management Layer
- Each repair replica detects and repairs failures from any component in the Management Layer (including repair replicas)

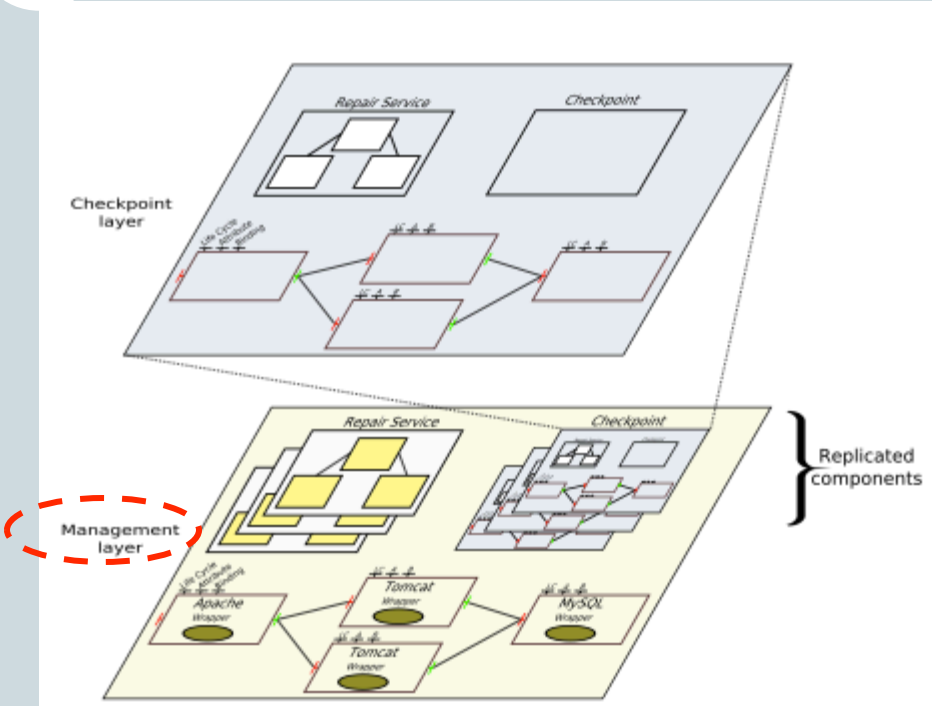


Figure 5: Putting pieces together: repair service & replicated components

Self-Healing Repair Algorithm

- Checkpoint-reference => reference to a checkpoint component
- Management-reference => reference to management component
- Algorithm 1 –
  - Step 1 – Algo 2
  - Step 2 – Algo 3
  - Step 3 – Algo 4
- Algorithm 2 –
  - Analyze the failure and return a repair plan according to the repair policy
  - Uses checkpoint layer to determine management state of the application prior to failure and then inserts reference of components to repair into plan

Algorithm 1 Global algorithm

Requires: An architecture-based management system satisfying the requirements exposed in this paper.

Ensures: The overall system is repaired in case of a node failure.

1: Analyse the failure and build a repair plan
2: Clean up the global system (remove failed components)
3: Execute the repair plan (substitute the failed components by newly ones)

Algorithm 2 Analyze the failure & build a repair plan

Requires: *FailedNode* : The checkpoint-reference of the component representing the failed node.

Ensures: *FailedCmps*: The management-references of the failed components. *RepairPlan*: A repair plan composed of the checkpoint-references of the components to repair (the default policy defines the components to repair as those that were running on the failed node)

1: **for all** *cmp* in *FailedNode.getSubComponents*() **do**
2: *FailedCmps.addCmp*(*managementRef*(*cmp*))
3: *RepairPlan.addCmp*(*cmp*)
4: **end for**

Self-Healing Repair Algorithm

- Algorithm 3 –
  - Locating all references from a surviving component to a failed component in the Management Layer
  - Removing those references
  - Uses meta-operations of the component model
  - Actions physically occur at Legacy Layer

Algorithm 3 Clean up the global system

Requires: *FailedCmps* : The management-references of the failed components.

Ensures: All relationships (binding, containment) involving a failed component are closed and removed.

```
1:  for all cmp in FailedCmps do
2:    {for each alive component, remove a failed binding}
3:    for all itf in cmp.getServerInterfaces() do
4:      for all clientItf in itf.getBindingSources() do
5:        if clientItf.owner() not in FailedCmp then
6:          clientItf.unbind()
7:        end if
8:      end for
9:    end for
      {for each alive component, remove a failed child}
10:   for all parentCmp in cmp.getParents() do
11:     if parentCmp not in FailedCmp then
12:       parentCmp.removeSubComponent(cmp)
13:     end if
14:   end for
      {for each alive component, remove a failed parent}
15:   for all subCmp in cmp.getSubComponents() do
16:     if subCmp not in FailedCmp then
17:       subCmp.removeParent(cmp)
18:     end if
19:   end for
20: end for
```

Self-Healing Repair Algorithm

- Algorithm 4 –
  - Algo 2 builds repair plan
  - Checkpoint-references of components to repair are in plan
  - Management state determined
  - Substitute failed component with new component
  - Modify state to same state of failed component prior to failure

Algorithm 4 Execute the repair plan

Requires: *RepairPlan* : a repair plan as returned by Algorithm 2, *FailedNode* : the checkpoint-reference of the component representing the failed node.

Ensures: The failed components are replaced by newly ones having the same management state

1: $newNode = NodeAllocator.replace(FailedNode)$
2: **for all** *cmp* in *RepairPlan* **do**
3: {Create an equivalent component (same attributes, interfaces, relationships and life cycle state) and deploy it on *newNode*}
4: **end for**

Experiment

- Paper came from ideas resulting from work with JADE architecture-based management system
  - JADE – (Java Agent DEvelopment Framework) - is a framework to develop multi-agent systems in compliance with the FIPA specifications.
  - FIPA specifications represent a collection of standards which are intended to promote the interoperation of heterogeneous agents and the services that they can represent.
- With work in JADE, leverage experience to prove model applied to a J2EE system; evaluate performance overhead and gain in availability
  - J2EE clustered web server
  - JMS message server
- In JADE Experiments used the FRACTAL reflective, Java-based component model, intended for construction of dynamically configurable and monitorable systems
  - FRACTAL, a hierarchical and reflective component model with sharing. Components in this model can be endowed with arbitrary reflective capabilities, from plain black-box objects to components that allow a fine-grained manipulation of their internal structure.
  - FRACTAL components have a reflective structure represented as:
    - Membrane – defines abstractions and its meta-level methods organized in *specializeable* controllers to provide the introspection and reconfiguration operations (i.e. Lifecycle State, Attributes and bindings)
    - Content – Internal attributes of component
  - Components enhanced with meta-data checkpointing facility and organized to support replication capabilities

Experiment

- Components enhanced with meta-data checkpointing facility and organized to support replication capabilities
- Two implementations of the management api; one for wrapping legacy components
- Table 1 shows specific and generic code size
  - A new administered legacy app would require a JADE wrapper (~360 lines avg Java code) plus FRACTAL configuration file (~20 lines of ADL)

J2EE Benchmark App

			# Java lines	# ADL lines
Generic code		Deployment Service	3505	1690
		Checkpoint layer	6630	–
		Replication layer	4567	832
		Self-Repair service	4750	430
		Total	***19452***	***2952***
Specific code	J2EE	Rubis app. - Web	150	11
		Rubis app. - Servlets	150	11
		Rubis app. - Database	150	11
		Total	***450***	***33***
		Apache Web server	800	16
		Tomcat Servlet container	550	12
		MySQL SGBD	760	40
		Total	***2110***	***68***
	JMS	JORAM server	368	51
		JNDI	134	12
		JMS Queue	253	16
		JMS topic	297	16
		Total	***1052***	***95***

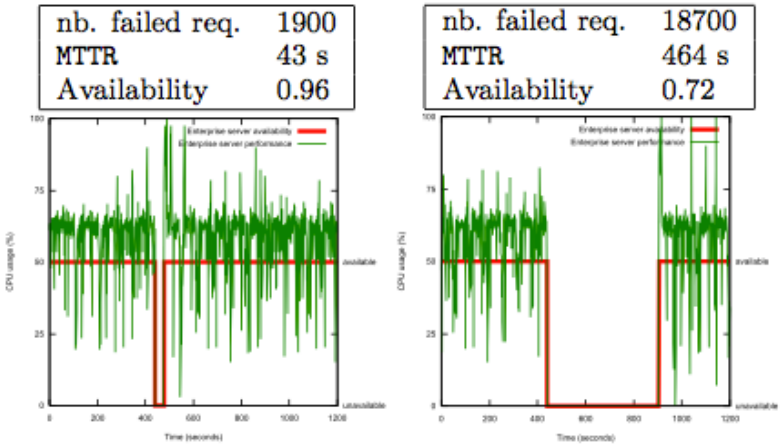
Table 1: Generic code vs. specific code

Experiment

- Performance overhead measures J2EE app with and without JADE components (repair service)
- Medium workload on small scale commodity hardware
- No failures induced so that execution under the control of JADE did not induce dynamic reconfigurations
- Figure 6 shows no significant overhead
- Figure 7 compares the availability of the RUBiS benchmark app with node failures – with human intervention vice JADE (autonomic)
  - Admin must detect error
  - Admin must know system architecture in detail
  - Admin then needs to isolate failure and know corrective actions
  - Admin then must implement them correctly
- In experiment – human was an expert waiting for a failure hence a minimal human MTTR expectation

	Human admin.	JADE admin.
Throughput	12 req./s	12 req./s
Resp. time	87 ms	89 ms
Mem. usage	17.5 %	20.1 %

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



(a) With JADE (b) Without JADE

Figure 7: Enterprise server behavior in the presence of failures

Related Work

- JMX – (Mbeans) – a quasi-standard for admin of Java apps i.e. J2EE App servers – limitation for this experiment is a lack of dependency exposure between managed objects (bindings & containment)
- CIM – Common Information Model) lacks uniform management interface
- SMARTFROG – framework for management of configuration-driven systems – only some attributes are specializable and lacks non-functional property definitions.
- Lira – lightweight infrastructure for managing dynamic reconfiguration – not clear how a repair manager could obtain necessary failure information and thus lacks ability to repair in generic fashion
- Architectural styles – authors believe approaches are complimentary to lower-level operators (i.e. JADE)
- Other Component models may lack attribute and life-cycle state functionality
- Middleware approach appears similar in component focus to this papers approach

Summary

- To build an autonomic repair service, the component model should:
  - (1) provide five main runtime abstractions:
    - Attributes
    - Interfaces
    - Life-cycle state
    - Binding and containment
  - (2) provide a way to manipulate these abstractions through specializable meta-operations (e.g., addSubComponent, bind, etc.).
- To enhance the repair service with self-healing behavior, the components have to support two non-functional properties:
  - A checkpointed property, allowing a component's meta-data to be checkpointed (e.g., containments, bindings, etc.),
  - A replicated property allowing components to be replicated without .
- Our experience has showed the importance of non-functional aspects of the component model such as meta-data checkpointing and replication. The soundness of our findings has been validated through several large scale experiments on successful middleware platforms. One is a clustered J2ee Web server and the other is a JMS message server.