Towards Autonomic Systems

Self-configuration and Self-repair

Sacha Krakowiak
Université Joseph Fourier, Grenoble
Project Sardes (LIG - INRIA)

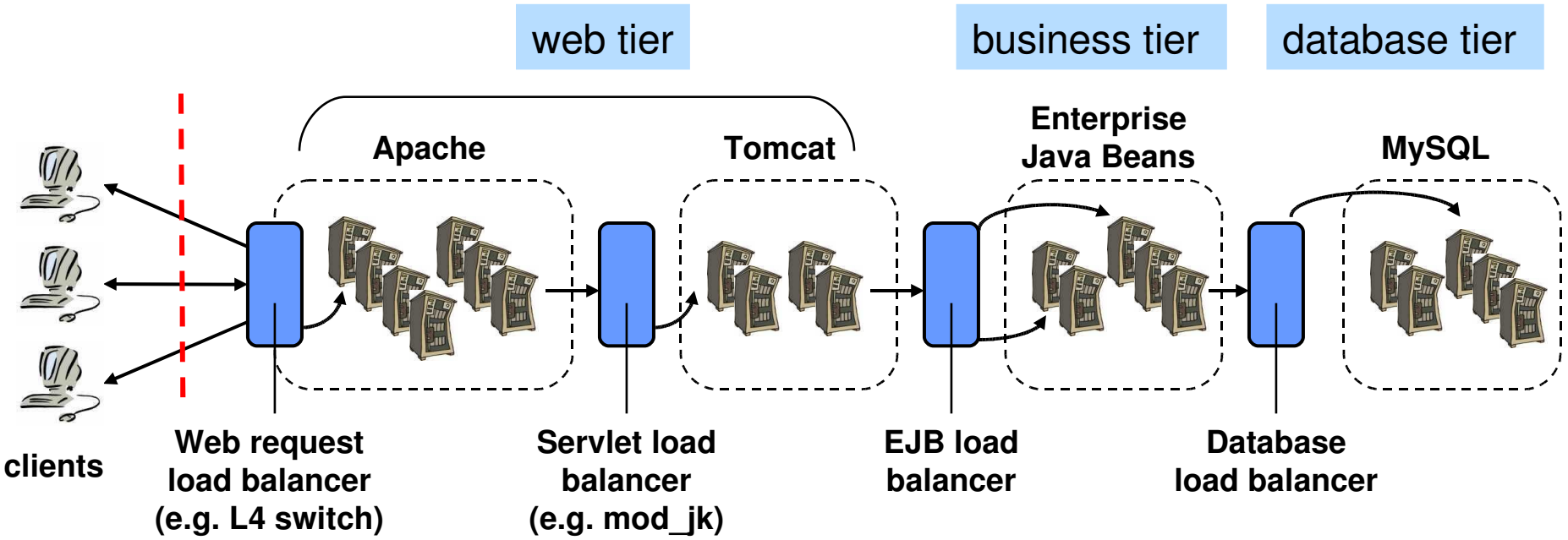
http://sardes.inrialpes.fr/~krakowia

The challenge of complexity

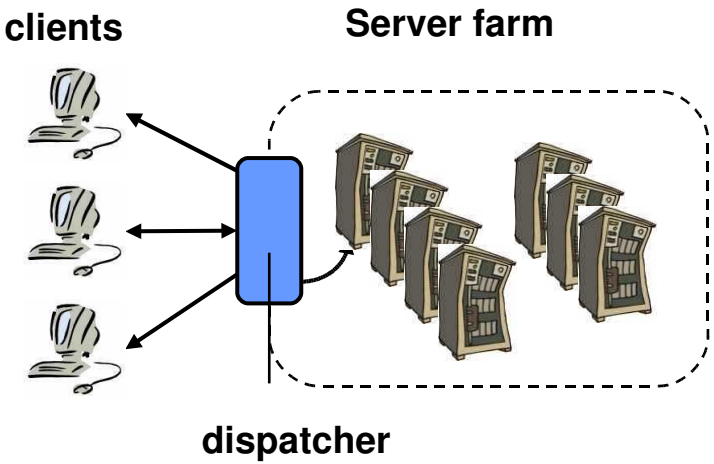
- An increasing number of human activities now rely on computing systems.
  - Communication, transportation
  - Commerce, finance
  - Energy production
  - Health care, ...
- **However, today's computing systems have become so complex that one hardly understands how they work...**
- **... and one hardly understands why they fail.**
  - **Some investigations**
    - **Gray (1985, 1989)**
    - **Murphy (1993)**
    - **Oppenheimer, Ganapathi, Patterson (2003)**

Examples of Internet services (1)

A multistage server (e.g., electronic commerce, etc.)

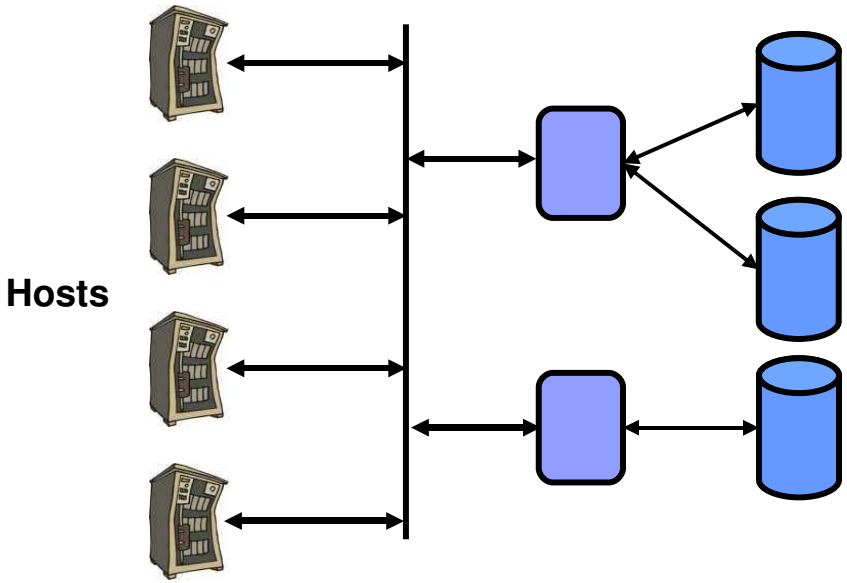


Examples of Internet services (2)



Using a cluster (or a grid) for CPU intensive computing

- scientific computing
- image synthesis and animation
- …



Managing a large scale storage utility for a set of customers

The origin of failures in Internet-based systems (1)

The failure data of three Internet systems have been analyzed and compared.
Characteristics of the services:

characteristic	*Online*	*ReadMostly*	*Content*
hits per day	~100 million	~100 million	~7 million
# of machines	~500 @ 2 sites	> 2000 @ 4 sites	~500 @ ~15 sites
front-end node architecture	custom s/w; Solaris on SPARC, x86	custom s/w; open-source OS on x86	custom s/w; open-source OS on x86;
back-end node architecture	Network Appliance filers	custom s/w; open-source OS on x86	custom s/w; open-source OS on x86
period studied	7 months	6 months	3 months
# component failures	296	N/A	205
# service failures	40	21	56

Component failure: failure of individual component (H/W, network, S/W, etc.)

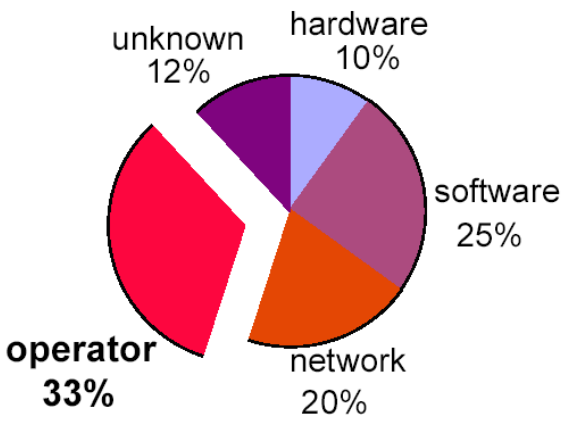
Service failure: failure visible to the end user (unmasked component failure)

D. Oppenheimer, A. Ganapathi, D. A. Patterson. Why do Internet services fail and what can be done about it? *Proc 4th Usenix Symp. On Internet Technologies and Systems (USITS'03)*, 2003

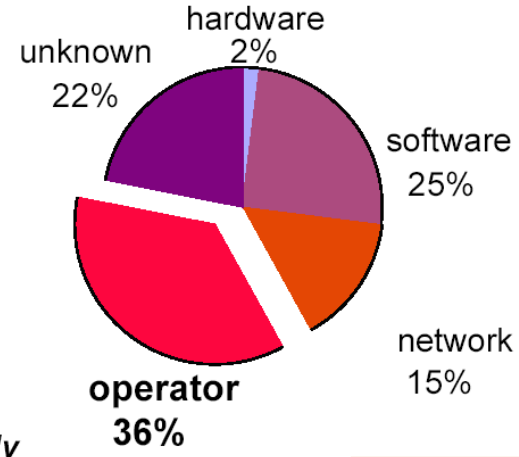
The origin of failures in Internet-based systems (2)

Failure cause by % of service failures

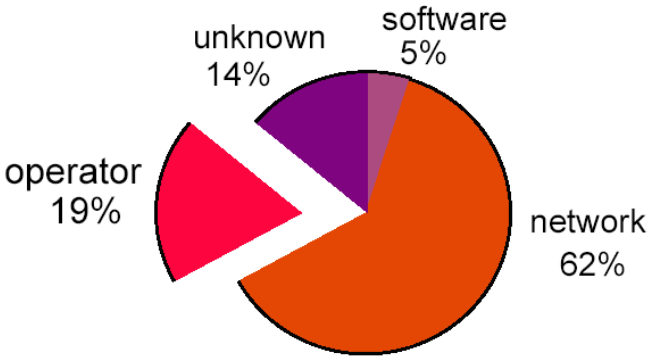
Online



Content



ReadMostly



Reminder:

A failure is a deviation from the specified behavior

A fault is any (potential) cause of a failure

In addition, most operator faults are configuration faults

Earlier studies:

Tandem Systems (Gray)
1985: Operator 42%, S/W 25%, H/W 18%
1989: Operator 15%, S/W 55%, H/W 14%

Vax (Murphy)
1993: Operator 50%, S/W 20%, H/W 10%

D. Oppenheimer, A. Ganapathi, D. A. Patterson. Why do Internet services fail and what can be done about it?
***Proc 4th Usenix Symp. On Internet Technologies and Systems (USITS'03)*, 2003**

Another symptom of complexity

- **IT spending in the 1980s**
  - 75% new hardware
  - 25% maintaining existing systems
- **IT spending in the 2000s**
  - 70-80% administering (repairing and maintaining) existing systems

The challenge of system administration

- System administration is getting too complex for humans
  - One remedy: computer-assisted administration

- **What is system administration?**
  - **Ensuring that the system provides a given level of quality of service**
  - **Maintaining this QoS level in the face of adverse conditions.**
- **Quality of service has many facets**
  - **Availability**
    - **Including partial availability**
  - **Performance**
    - **Mean throughput, latency, etc.**
    - **Differentiated levels**
  - **Security**
    - **Well-known and new threats**

System administration tasks

- Defining policies
  - Defining QoS evaluation criteria
  - Defining goals
  - Setting priorities
- **Configuring a system**
  - **Selecting components**
  - **Choosing location for placement**
  - **Setting parameter values**
- **Reacting to external events**
  - **Hardware, software or network failure**
  - **Load peak**
  - **Security attack**

Can be (partially) automated

Towards autonomic computing

- **A concern for the industry**
  - The *Autonomic Computing* initiative (IBM)
  - The *Adaptive Enterprise* (HP)
  - The *Dynamic Systems* initiative (Microsoft)
  - ... others
- **A long term goal: self-managing (or autonomic) systems**
  - Self-configuration
  - Self-optimization
  - Self-repair
  - Self-protection
- **Complete automatic self-management will probably not be achieved**

Plan of this talk

- An introduction to autonomic computing
  - Concepts, techniques and frameworks
  - Architecture-based system management
- Case studies
  - The Jade framework for autonomic computing
    - Self-configuration
    - Self-repair
- A word of conclusion

Reacting to change

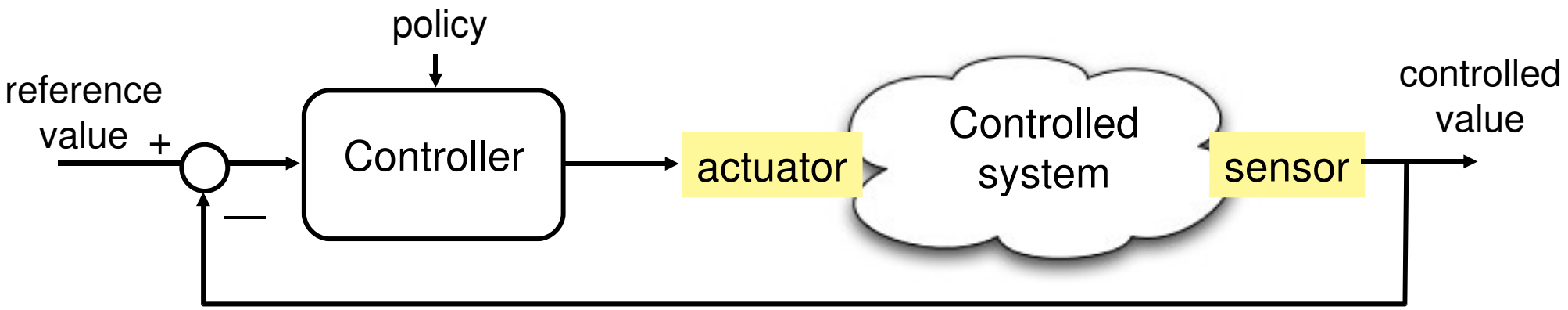
- Computing systems operate in a changing environment
  - User needs (QoS)
    - Performance
    - Availability
    - Security
    - Differentiated QoS
  - Resource availability
    - Hardware
    - Software
    - Network
  - External events
    - Load peak
    - Faults (hardware, software, operator)
    - Security attacks (penetration, denial of service)
- Adaptation mechanisms are the basis of autonomic computing
  - Two main approaches

Approaches to adaptation

- **Two main approaches**
  - **Self-awareness (control-based)**
    - **The system maintains a model of itself (reflection), and of its environment**
    - **The system is regulated through explicit control (reasoning and planning, based on the model)**
    - **Main approach in industry - also active research topic**
  - **Self-organization (cooperation-based)**
    - **The system is regulated by the cooperating behavior of its elements**
    - **Analogy: biological and social systems**
    - **Investigated in research projects (“ant colonies”, etc.)**

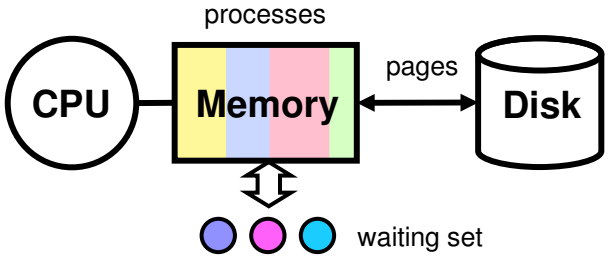
We follow the self-awareness approach in this talk

Feedback control: A quick refresher

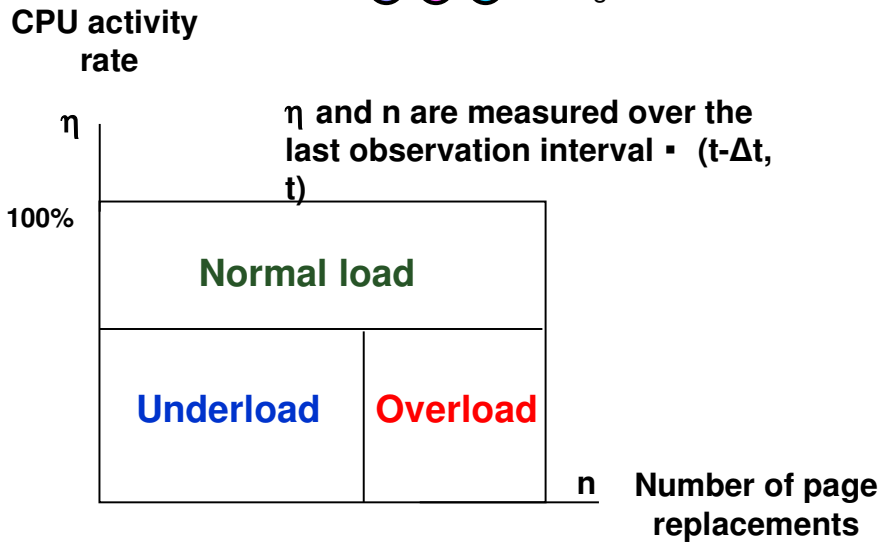


- **Goal of control**
  - **To influence the behavior of a system, in order to make it conformant to (or to reduce the discrepancy with) a prescribed scheme**
    - **Many examples in everyday life and industry (thermostat, automatic pilot, chemical processes, etc.)**
- Why is it difficult?
  - The input to the controlled system may be unknown or unpredictable
  - The controlled system may be subject to unpredictable perturbations (noise, etc)
  - The interaction with the controlled system may be imperfect
  - The controlled system may be very complex, and its behavior not completely understood

An early example of control in computing systems



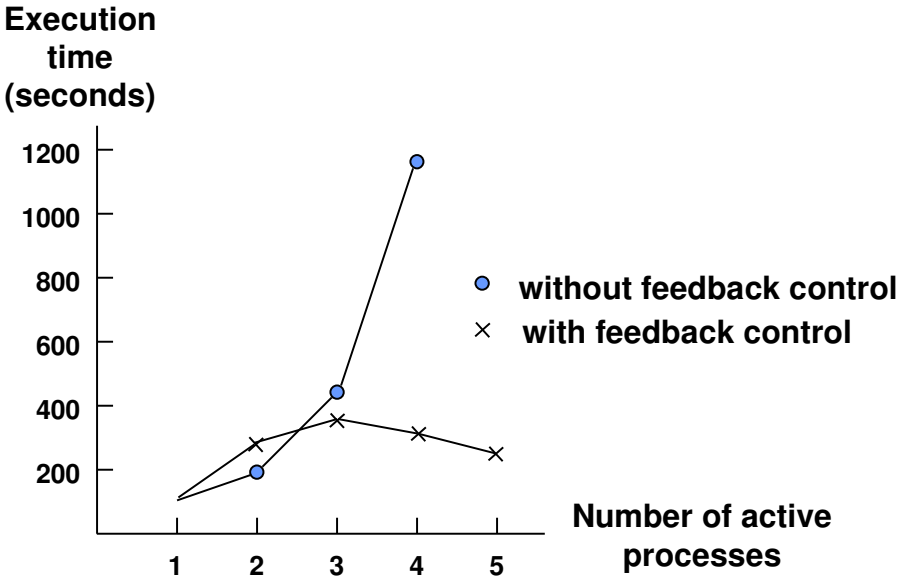
Preventing thrashing: the IBM M44/44X experiments (1968)



a) a simple model of system behavior

```
every Δt do
      if  (overload)
            move one process from ready set to waiting set
      else
            if (underload and (waiting set ≠ ∅))
                  admit one waiting process to ready set
```

B. Brawn, F. Gustavson. Program behavior in a paging environment. *Proc. AFIPS FJCC*, pp. 1019-1032 (1968)



b) the effect of load limitation by admission control

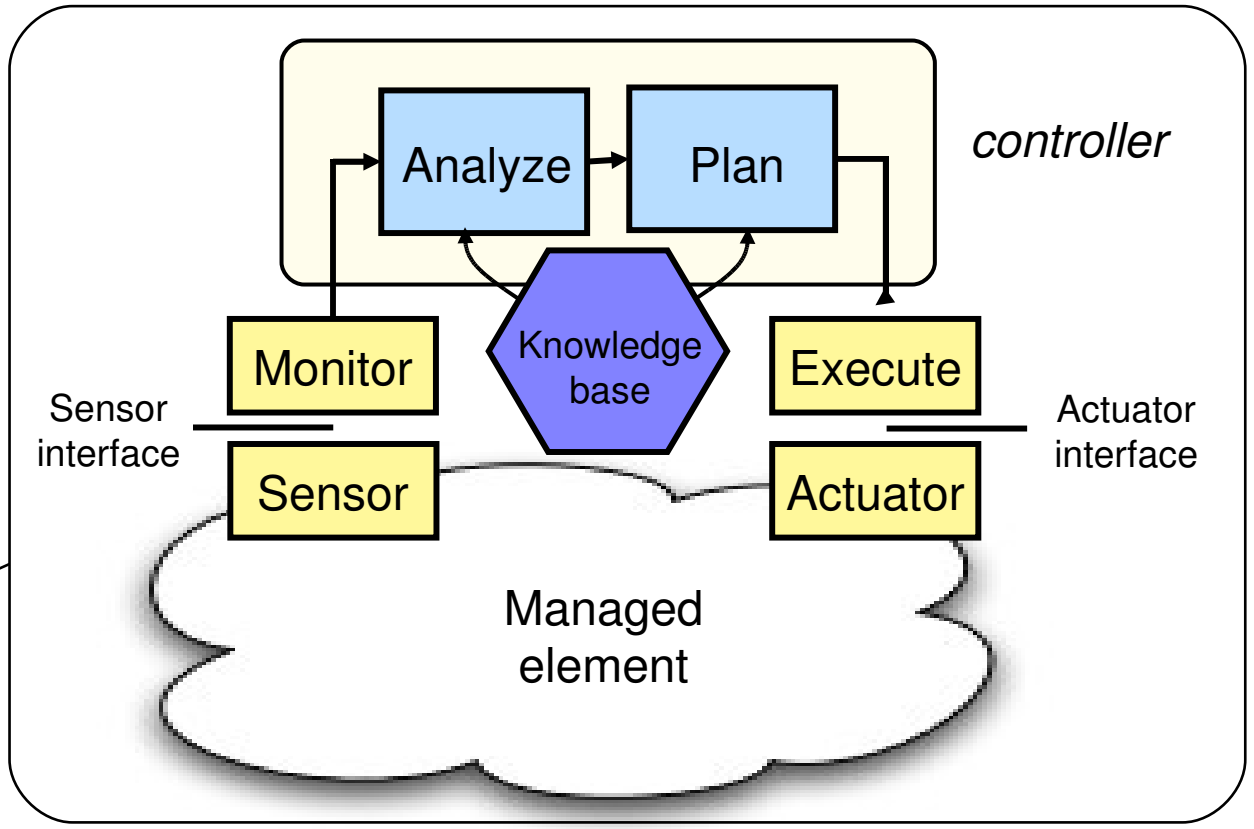
Applying control to computing systems

- **What is different?**
  - Large scale (large number of elements, complex interactions)
  - Behavior highly non-linear (thresholds, etc.)
  - Output variables difficult to obtain
    - QoS difficult to measure
    - Often approximated by resource utilization
- **A bit of history**
  - In the 1970s: some attempts, limited success
  - In the 1980s: networks, adaptive routing
  - In the 1990s: QoS for networks, open-loop scheduling (fair share, etc.)
  - In the 2000s: Autonomic computing for Internet services, model-based approaches

First book on the subject: Hellerstein et al., *Feedback Control of Computing Systems*, Wiley 2004

A basic framework for autonomic computing

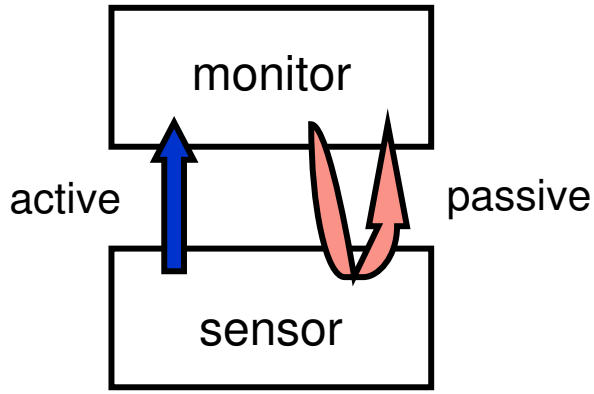
The MAPE-K framework
(as originally proposed by IBM)



- **A system is made up of autonomic elements**
  - **Each element is individually controlled**
  - **A form of global control is needed**

Sensors

- **What to measure?**
  - **Resource occupation (CPU or memory consumption, number of calls, ...)**
  - **End user performance (latency, throughput)**
  - **Failure occurrence (hardware, software)**
  - **Erratic behavior (deviations from specifications)**
- **Active vs passive**
  - **Active: triggered by sensor**
  - **Passive: triggered by monitor**
  - **Example: *heartbeat* vs *ping* for failure detection**



- Raw vs synthetic
  - Probes: deliver direct measurement results
  - Gauges: analyze data delivered by probes to deliver higher level view
- Local vs aggregated
  - Mining/aggregating data over a large system
  - Collecting statistics
  - Hierarchical organization (domains)

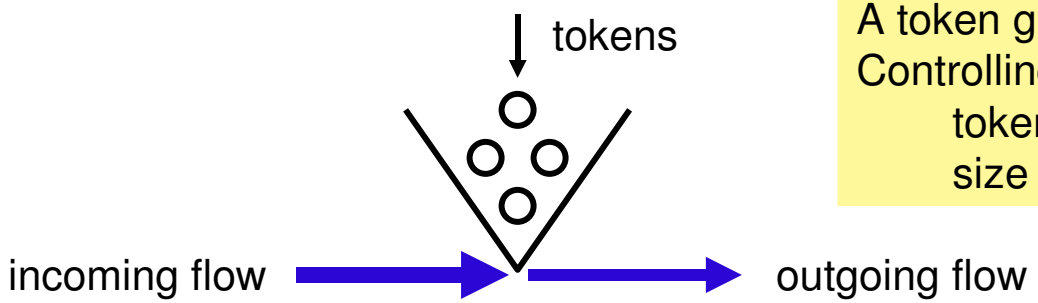
Action modes

- **Using predefined knobs**
  - Example: local reboot
- **Controlling the load: admission control**
  - A general technique for resource management
  - Idea: admit additional load only if it does not overload the system. Assumes
    - reasonable estimates of current and additional load
    - reasonable capacity planning and provisioning
  - Non-intrusive
- **Controlling resource usage**
  - Request scheduling
  - Resource (re)allocation and sharing
  - Assumes access to resource management
- **Changing the system's structure**
  - Dynamic reconfiguration
  - Assumes some kind of modular structure

Actuators

Actuators for admission control

- Individually accept/reject requests
  - May involve estimated cost of rejection
- Flow throttling
  - Example: token bucket



A token gives a right to a unit of flow (T bit/s).
Controlling variables:
token issue rate
size of bucket

Actuators for reconfiguration

- No accepted standard yet
- A whole area of research
- A component-based organization greatly helps

Approaches to autonomic control

- **Empirical**
  - Does not need detailed knowledge of the managed system
  - Often driven by “event-condition-action”
- **“Black box” model**
  - Assume some behavior law (e.g., linear, time-invariant, etc.)
  - Determine model parameters by identification
  - Use model to determine response, using control theory
- **“Queueing” model**
  - Represent system by network of queues
  - Use queueing theory to determine response
  - Mainly used for performance evaluation and capacity planning
- **Specific model**
  - Needs detailed knowledge of the managed system
  - Limited to fairly simple subsystems (e.g., task scheduling)

What comes next …

- The main areas of autonomic computing
  - Self-protection
  - Self-optimization
  - Self-configuration
  - Self-repair (or self-healing)
- Some current approaches
  - Architecture-based management
  - Reflective systems
- Case study

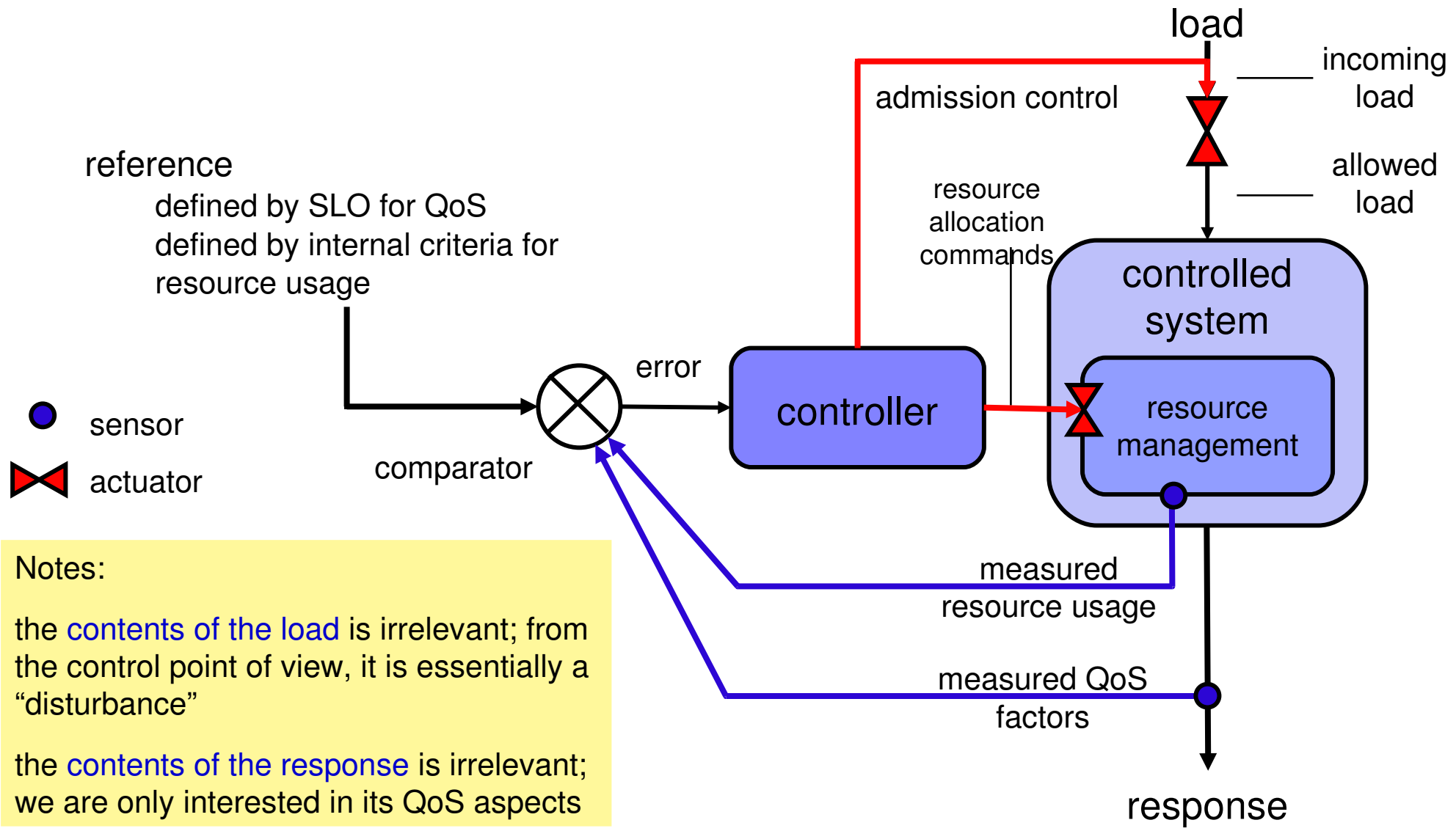
Self-protection

- **Motivation**
  - Protect the system against malicious attacks
- **Goal**
  - Prevent intrusion
  - If the attack succeeded
    - Repair its effects
    - Make the system resistant to further attacks
- **Problems**
  - Security holes are unavoidable
  - New forms of attacks are invented continuously
  - Intrusion detection is hard (and when succeeds, much damage has been done)
  - Human expertise needed (for the time being)
- **Approaches**
  - Specify “normal” behavior
  - Define “sense of self” (biological analogy)

Self-optimization

- **Motivation**
  - Respect the Service Level Agreement in a changing environment
- **Goal**
  - Keep the user-perceived performance at an acceptable level, in spite of unexpected events (peak load, resource unavailability, etc.)
  - Maintain fairness (with possibly different levels of service)
  - Optimize resource usage
- **Problems**
  - Measuring user-related QoS factors is difficult; complex correlation with resource usage; sometimes contradictory goals
  - An accurate model of the system is seldom available (“black box”)
  - The behavior of the system is highly non-linear (threshold effects, etc.)
- **Approaches**
  - A general heuristic: admission control
  - Model-based or empirical feedback control

Self-optimization: the big picture

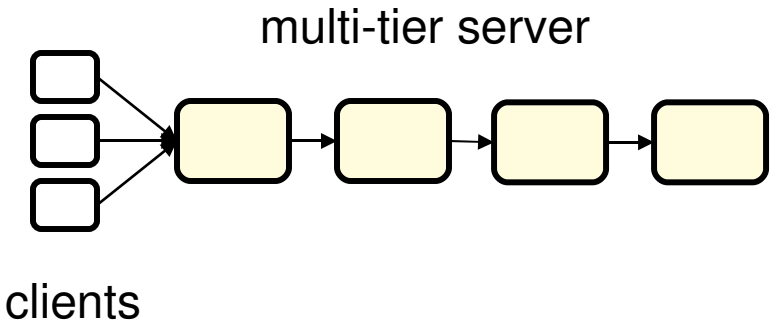


Notes:

the contents of the load is irrelevant; from the control point of view, it is essentially a "disturbance"

the contents of the response is irrelevant; we are only interested in its QoS aspects

Example: self-optimization of a web server



Goal

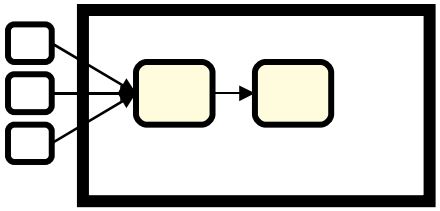
Avoid thrashing) in the presence of overload: maintain high throughput, low response time

Desirable properties

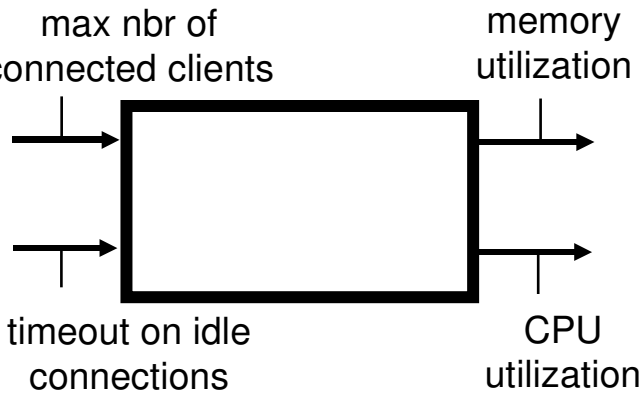
- Apply to dynamic content services (response to request is dynamically generated)
- Use non-invasive techniques (keep server unmodified)
- Control user-related (instead of server-related) variables

Self-optimization of a web server: example 1

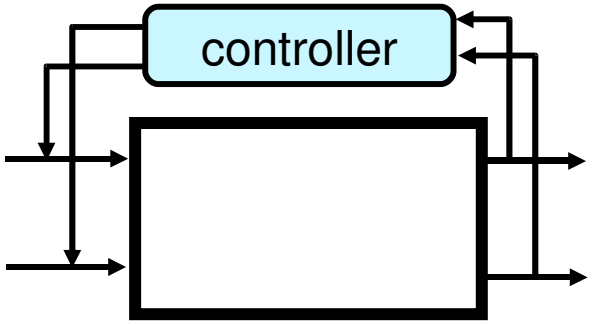
Consider the server as a black box with linear behavior; use identification techniques to determine parameters; apply feedback control techniques (MIMO).



clients



determine parameters by identification
(assume linear model)



design controller

This was a pioneering experiment; however, it had limitations:

- Only applied to static content (Apache web server)
- Observed parameters were server-related (not end user performance)
- Limited to linear model

Y. Diao, N. Gandhi, J. L. Hellerstein, S. Parekh, D. M. Tilbury. Using MIMO Feedback Control to Enforce Policies for Interrelated Metrics with Application to the Apache Web Server, *Proc. Network Operations & Management Symp. (NOMS'02)*, Florence, April 15-19 2002

Self-optimization of a web server: example 2

Gatekeeper: measurement-based admission control at DB tier, with request scheduling.

Needs a preliminary measurement step to determine capacity

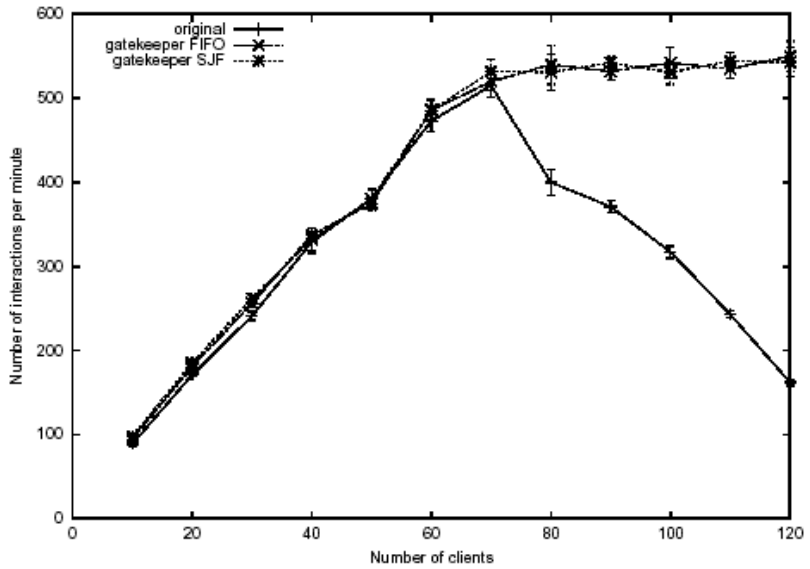
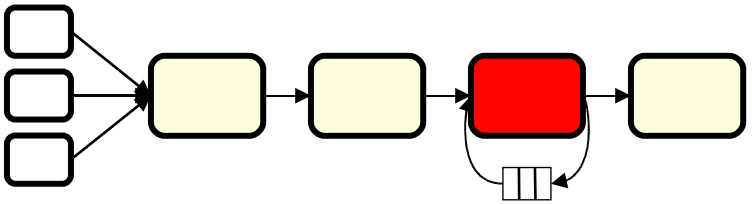


Figure 7: Throughput (MySQL, locking in database).



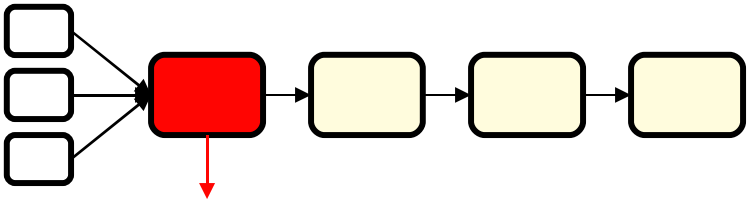
Elnikety, Sameh, Nahum, Erich, Tracey, John and Zwaenepoel, Willy (2004) A Method for Transparent Admission Control and Request Scheduling in E-Commerce Web Sites. In *Proceedings International WWW Conference*, New York, USA.

Limits

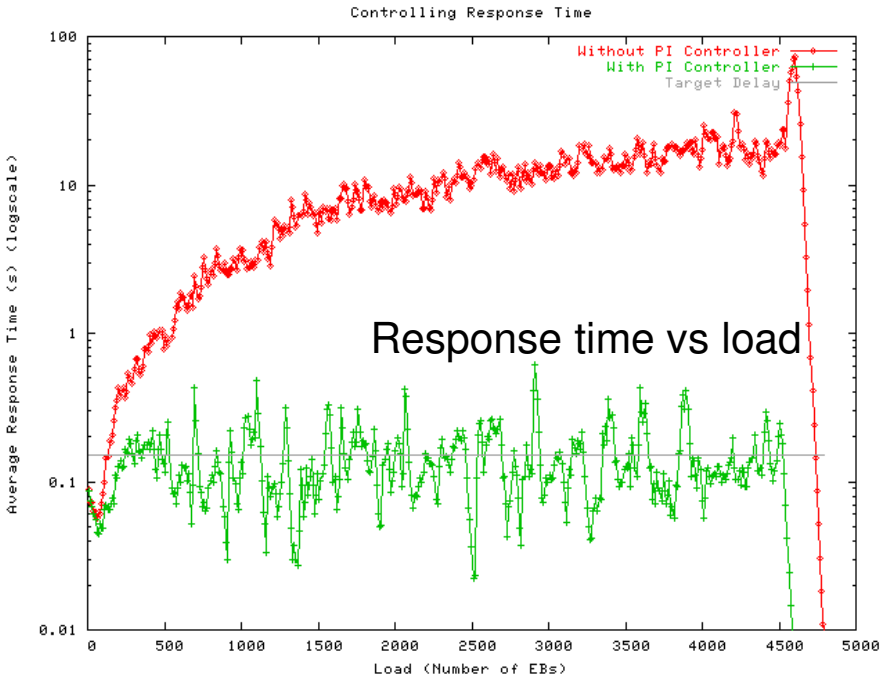
- single machine server
- artificial load generation
- preliminary calibration necessary

Self-optimization of a web server: example 3

Yaksha: admission control with adaptive PI feedback. The parameters of the model are dynamically adjusted. Simple implementation (proxy)



Abhinav Kamra, Vishal Misra, and Erich Nahum. *Yaksha: A self-tuning controller for managing the performance of 3-tiered web sites*. In International Workshop on Quality of Service (IWQoS), Montreal, June 2004.

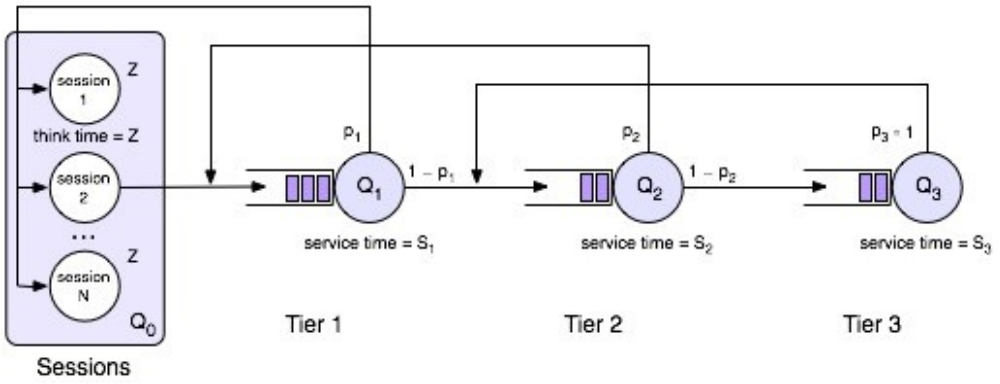


Limits

- single machine server
- artificial load generation
- model needs refinement (interaction between tiers, etc.)

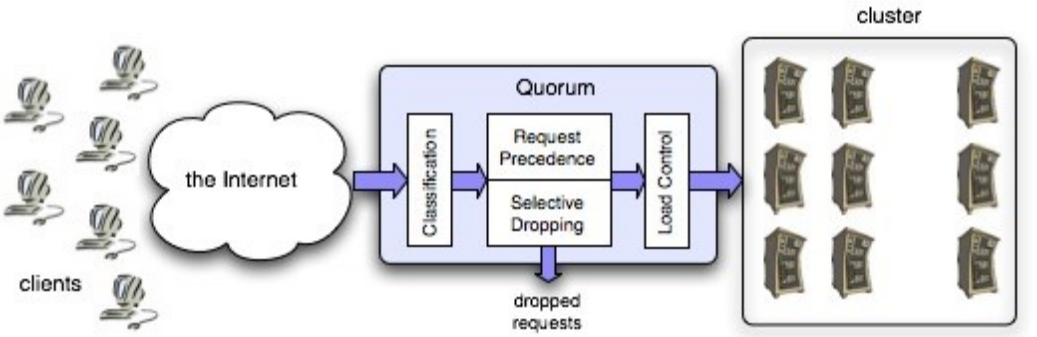
Self-optimization of a web server: other approaches

A complete queueing model for a multi-tier service, allowing multiple servers and multiple service classes. Mainly used for capacity planning and dynamic resource provisioning for predictable loads



Urgaonkar, B., Pacifici, G., Shenoy, P., Spreitzer, M., and Tantawi, A. (2007). An Analytical Model for Multi-tier Internet Services and its Applications. ACM Transactions on the Web, 1(1).

A front-end "black box" for admission control to a clustered server, ensuring QoS guarantees for multiple service classes, again for predictable loads



Blanquer, J. M., Batchelli, A., Schauser, K., and Wolski, R. (2005). Quorum: Flexible Quality of Service for Internet Services. In Second USENIX Symp. on Networked Systems Design and Implementation (NSDI'05), pages 159-174, Boston, Mass., USA.

Self-(re)configuration

- **Motivation**
  - Configuration is a tedious, complex, error-prone process (see failure statistics)
- **Goal**
  - Starting from a high-level description of a system's organization, deploy and start a working instance of the system
  - Change the structure and/or the composition of a working system, according to specified rules
- **Problems**
  - How to describe the organization and constraints of the system?
  - How to translate rules into actions?
  - How to "package" the system?
- **Approaches** (see case study)
  - Architecture-based description (possibly model-based)
  - Automatic generation of tasks for a workflow engine

Note that (re)configuration is both a primary objective **and** an actuator for other aspects

Self-repair

- **Motivation**
  - Maintain the system's availability
- **Goal**
  - Suppress or minimize the (user perceived) effects of a failure
- **Problems**
  - Many failures (specially in communication) do not follow the fail-stop mode
  - Tracing the precise location of a software failure is difficult
  - Restoring state is a complex issue
- **Approaches**
  - Relate failure to system structure: architecture-based approach (see case study)
  - Reduce recovery time
    - Early detection
    - Fast restoration (example: Micro-reboot, after fine-grained location)
  - Consider degraded mode operation (not all failures are fatal)
    - Performability studies (fault injection, etc.)

Current approaches to configuration and repair

- **Model-based policies**
  - Expressing management policies in terms of high-level models of the managed systems
- **Architecture-based management**
  - Using the architecture of the managed system as a guide for developing autonomic behavior
- **Reflective systems**
  - Using reflection as a basic tool for implementing sensors and actuators

Architecture-based management

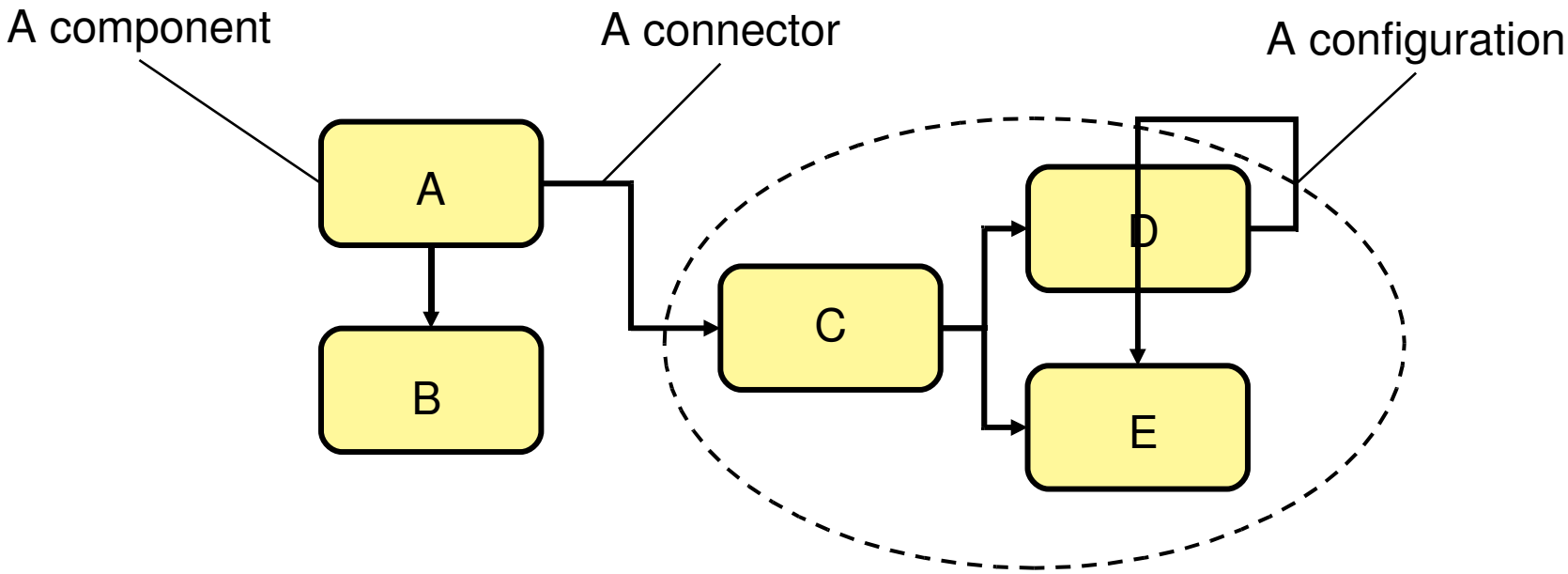
System architecture

- A framework for describing a system as an assembly of parts
- Basic notions: component, connector, configuration
- High-level global description: Architecture Description Languages (ADL)

Why architecture-based management?

- Management notions conveniently map on architectural notions
- Components are units of deployment, fault diagnosis and isolation, domains of trust
- Reconfiguration is represented by component replacement and connector rebinding
- Components can be equipped with standardized interfaces for sensors and actuators
- Therefore components are a convenient base for managed elements

Main concepts of system architecture



A (simplistic) architecture description

```
component A, B, C, D, E
configuration X exports C
                C uses D, E
                D uses E
A uses B, X
```

Other notions: interface, conformance, meta-data, etc. (to be illustrated in case study)

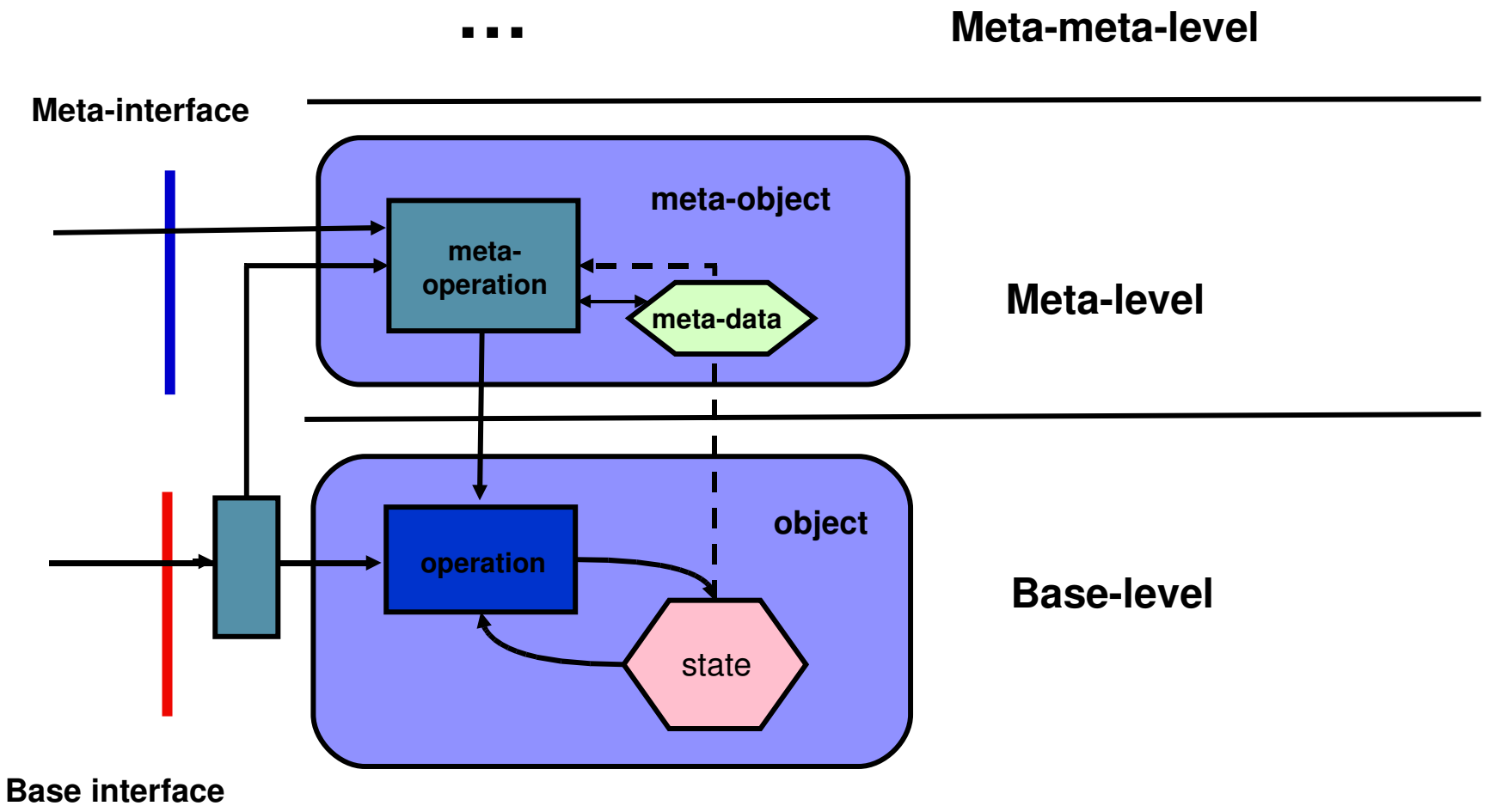
Some current issues in systems architecture

- **Describing architectures (ADL)**
  - No standard language currently
  - Trend: core language + extensions
  - XML-based for convenience and standardization
- **Managing architectural change**
  - Describing “architectural diffs”
  - Performing dynamic changes
    - Defining consistency criteria
    - Expressing change (“dynamic ADL”)
    - From expression to action

Introducing reflection (1)

- **Reflective systems**
  - A reflective system: one that maintains a representation of itself
  - The meta-level gives a concrete form to the system state (reification)
  - This allows the system to be inspected or modified through a meta-level interface
  - The representation needs to be causally connected to the system
- **Using reflection for system management**
  - The meta-level interface is a convenient interface both for sensors (introspection) and for actuators (intercession)
- **Reflective component models**
  - The meta—level interface reifies architectural properties (life cycle, connections, etc.)

Introducing reflection (2)



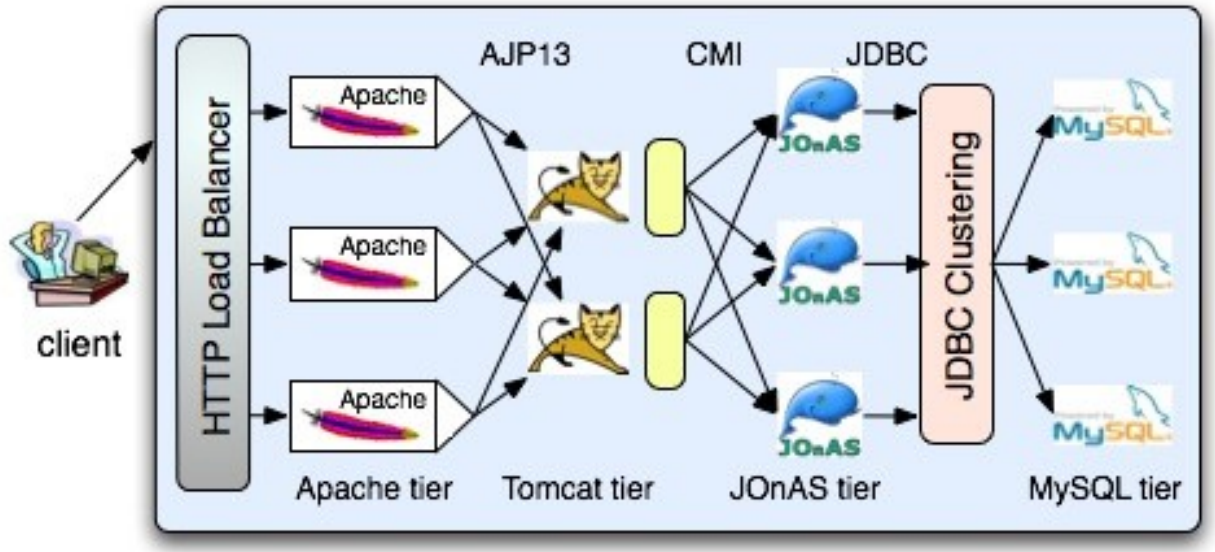
Jade, an experiment in architecture-based self-management

The Jade project

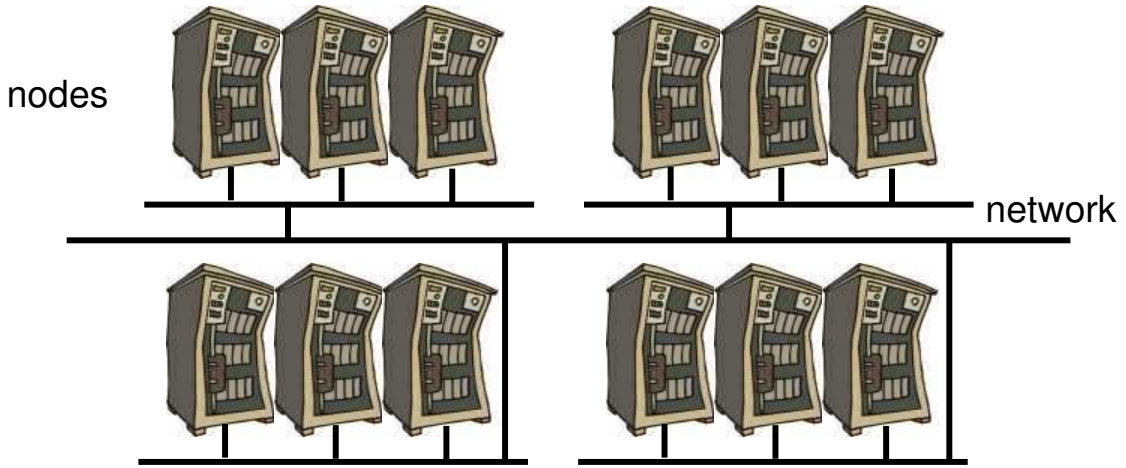
- Developed by research team Sardes (University of Grenoble and INRIA, 2003-2008)
- A framework based on reflective components
- Experiments in various aspects of autonomic computing (configuration, performance, security, fault tolerance)
- Targeted to medium to large size clusters for Internet services
- One industrial application (with Bull)
- Site: http://sardes.inrialpes.fr/jade.html
- Recent publication:
  - S. Sicard, F. Boyer, N. De Palma. Using Components for Architecture-based Management: the Self-repair Case, *Proc. International Conference on Software Engineering (ICSE 2008)*, Leipzig, Germany
- The following presentation is mainly based on that paper
  - Thanks to the authors

The managed system: JEE server

The application software



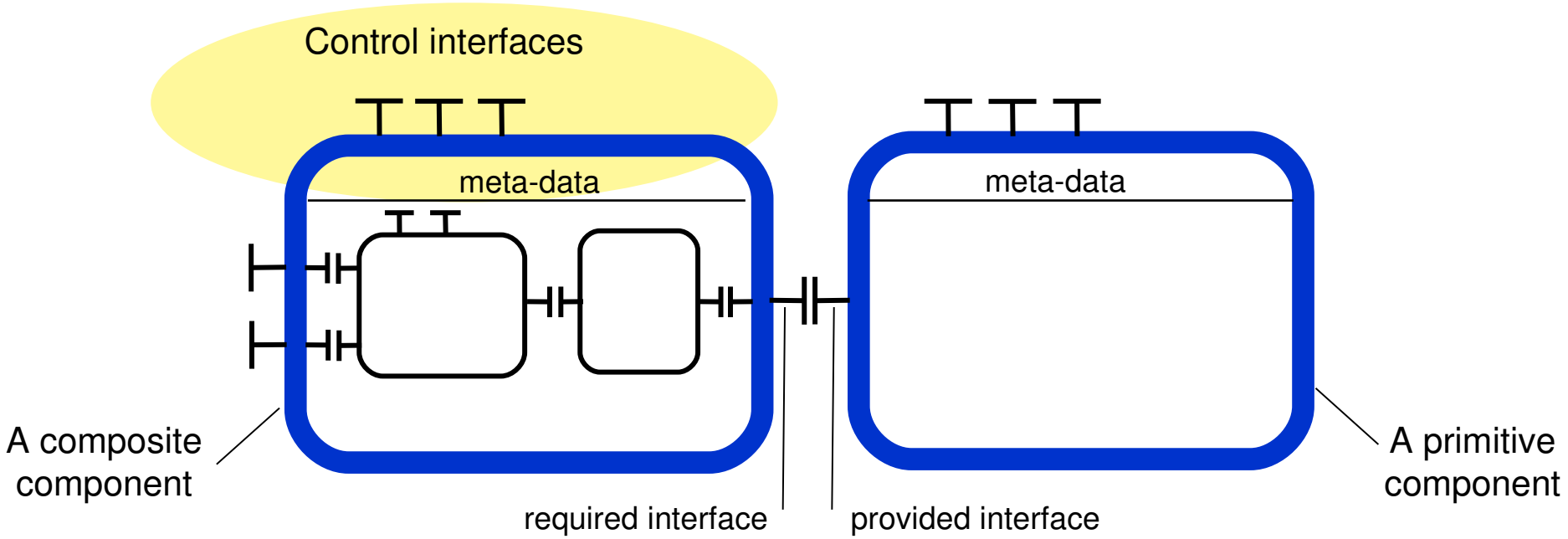
The hardware infrastructure



Fractal, a reflective component model

- **Main features**
  - **A general component model, allows hierarchical composition and sharing**
  - **Three sorts of interfaces: provided, required, and control (meta—level)**
  - **Components are run time structures**
  - **High—level architectural description through an ADL**

- The meta—level interface
  - Attribute controller: read/modify the state variables
  - Life cycle controller: start, stop
  - Binding controller: manages connections
  - Contents controller: manages included components
  - This list is optional and extensible

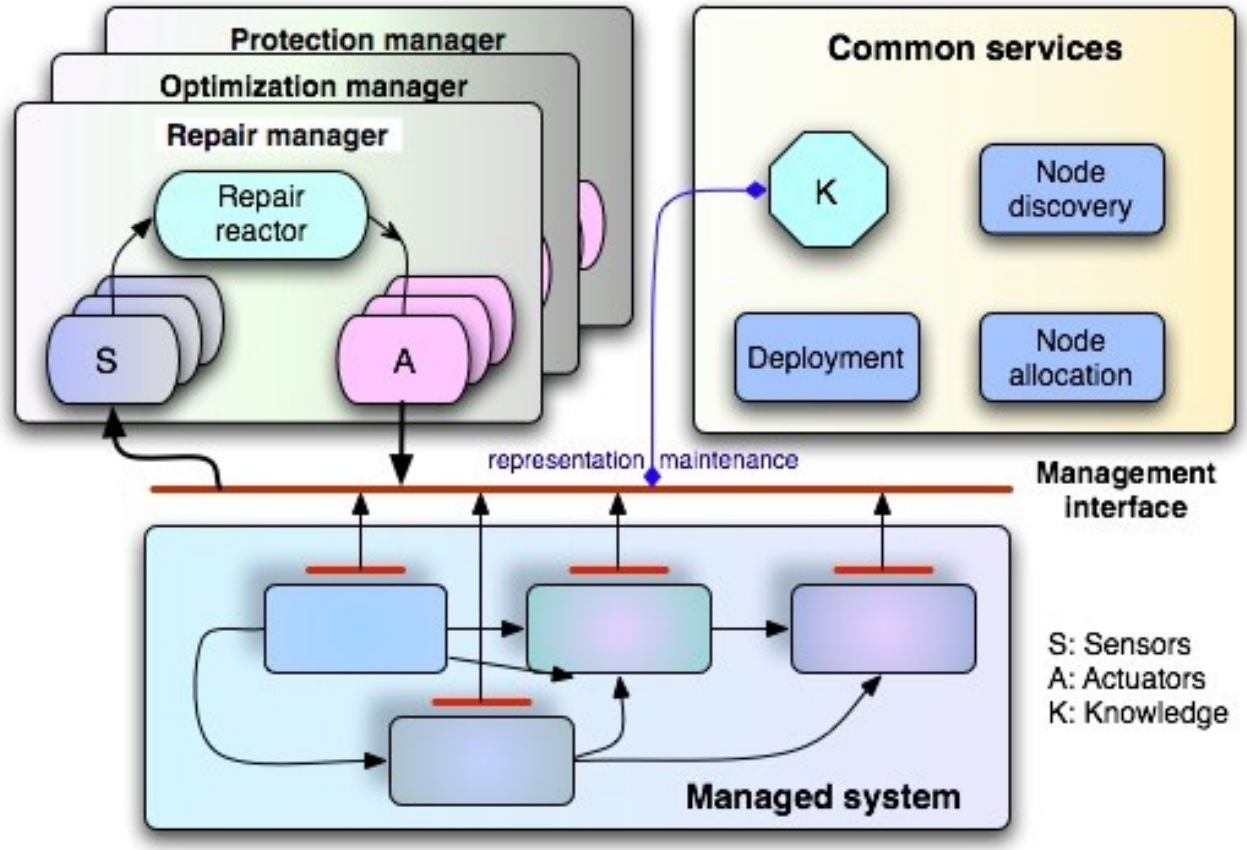


An overview of Jade

Both the managed system and Jade itself are organized as as an assembly of Fractal components.

To manage legacy systems, one needs to wrap them into Fractal components.

The architecture of the managed system is described in Fractal ADL



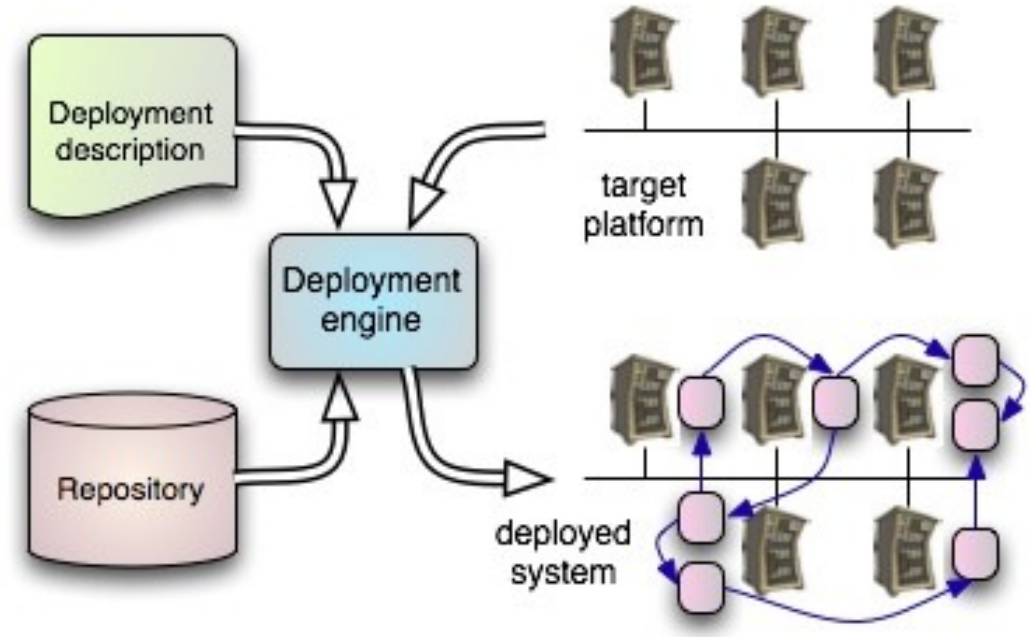
Using Fractal in Jade

- To describe the managed application
  - An assembly of components
  - Each component has a management (meta-level) interface
  - Using wrappers for legacy applications
- To describe the hardware platform
  - The platform is a set of virtual nodes
  - A virtual node is a wrapped physical node
    - Provides a Fractal-style management interface
  - Virtual nodes are mapped to physical nodes
    - Node allocation service
- To describe the Jade system
  - …using Jade to manage itself
    - Example: self-repairing repair system

The Jade deployment service

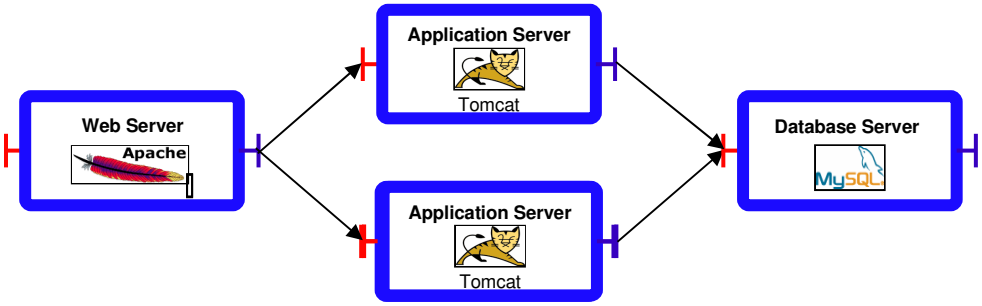
Deployment is the process of making a system ready for use by installing it on a platform

The system is described in a high-level Architecture Description Language (ADL)



Describing a system configuration

A J2EE 3-tier application




```
-- ====  <!-- MYSQL                                -->
<!-- TOI  <!-- ====================================    -->
<!-- ===  <component name="mysql"
<compc  definition="fr.jade.resource.j2ee.mysql.MysqlResourceType">
    defin      <attributes>
    <attr          <attribute name="resourceName" value="mysql" />
       <           <attribute name="dirLocal" value="/tmp/j2ee" />
       <           <attribute name="user" value="jlegrand" />
       <       </attributes>
       <       <virtual-node name="node1" />
    </att      <packages>
    <virt          <package name="MySql (linux x86)" />
    <pac           <package name="MySql Wrapper" />
       <       </packages>
      <p   </component>
    </pa   <!-- ====================================    -->
</compo  <!-- BINDINGS                              -->
<compc  <!-- ====================================    -->
    defin      <binding client="apache.worker1" server="tomcat1.resource" />
    <attr      <binding client="apache.worker2" server="tomcat2.resource" />
       <       <binding client="tomcat1.jdbc" server="mysql.resource" />
       <       <binding client="tomcat2.jdbc" server="mysql.resource" />
       <       <virtual-node name="node1" />
       <   </definition>
    </att
    <virt
    <pac
        <package name="Tomcat (linux x86)" />
        <package name="Tomcat Wrapper" />
    </packages>
</component><!--
```

Practical aspects of deployment (1)

- **Re-engineering a legacy system in component form**
  - Technique: wrapping parts of the system into Fractal components
  - Example: a J2EE server (JOnAS, an open-source implementation)
- **Packaging the components for deployment**
  - A package is a unit of independent deployment
  - In Jade, we use the OSGi standard (“bundles”)
    - Independent units of execution
    - Run-time isolation (for reconfiguration)
    - Possible coexistence of multiple versions
  - Two kinds of bundles
    - Component bundles (contain implementation + needed Java services)
    - Interface bundles (contain Java interfaces needed for binding)
    - This allows removing direct inter-component dependencies (they only depend on interfaces)

Practical aspects of deployment (2)

- Implementing the deployment engine
  - Extract information on target machine
  - For each component:
    - find component's code and package identifier
    - Call Generic Factory on target machine
      - Generic Factory calls Installer
      - Installer downloads package from repository
      - Installer asks LoaderFactory to create loader for the package
      - Generic Factory uses loader to create an instance of component
    - Configure component through control interface
  - Set bindings between components
    - Use ADL description
    - Call binding controllers
  - Start configuration
    - Call lifecycle controllers

Dynamic reconfiguration (1)

Dynamic reconfiguration is the process of changing the configuration of a deployed system, at run time

Reconfiguration is one actuator used by autonomic systems

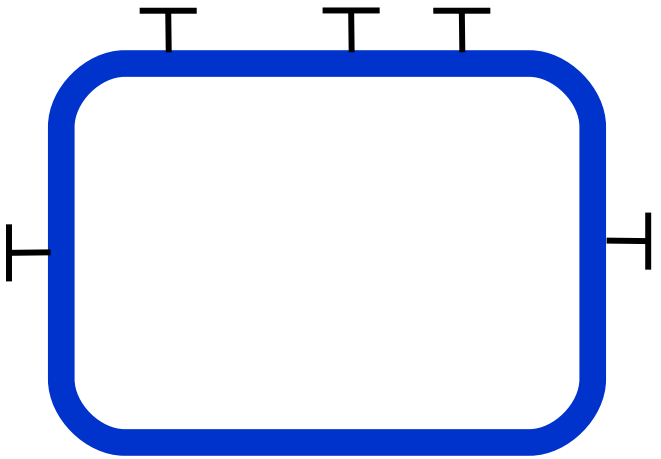
Reconfiguration is done through the meta-operations of components; a reconfiguration script may be generated from a dynamic ADL program

Examples

binding controller: add/remove a binding

content controller: create a contained component

lifecycle controller: start/stop the component



Dynamic reconfiguration (2)

Objective: change an application's configuration (e.g., replacing some components) without stopping the application

Method:

temporarily suspend the execution of the modified part
do the modification
resume execution

Problems:

limit the extent of the suspended part
preserve consistency
… subject of ongoing research

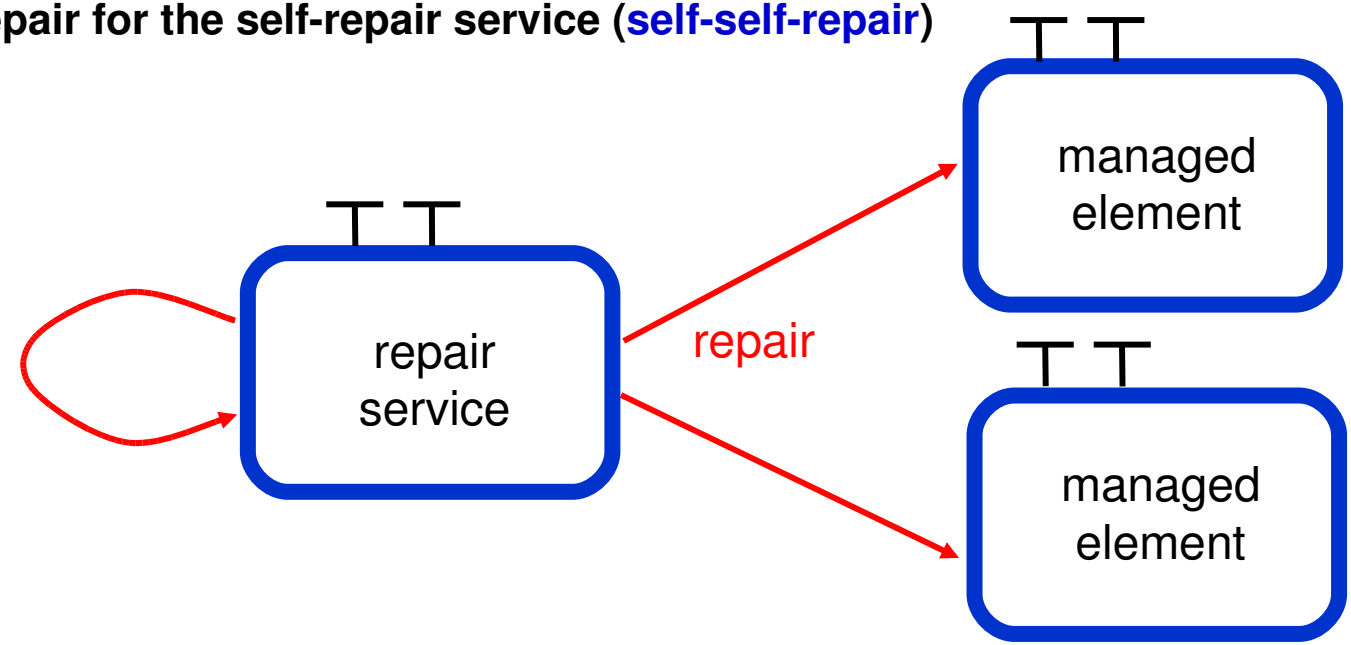
Implementation:

Use the `LifeCycleController` Fractal interface
Methods

`stopFc`: suspend execution of activities within (composite) component
`startFc`: resume the execution suspended by `stopFc`
`getFcState`: know the current execution state of the component

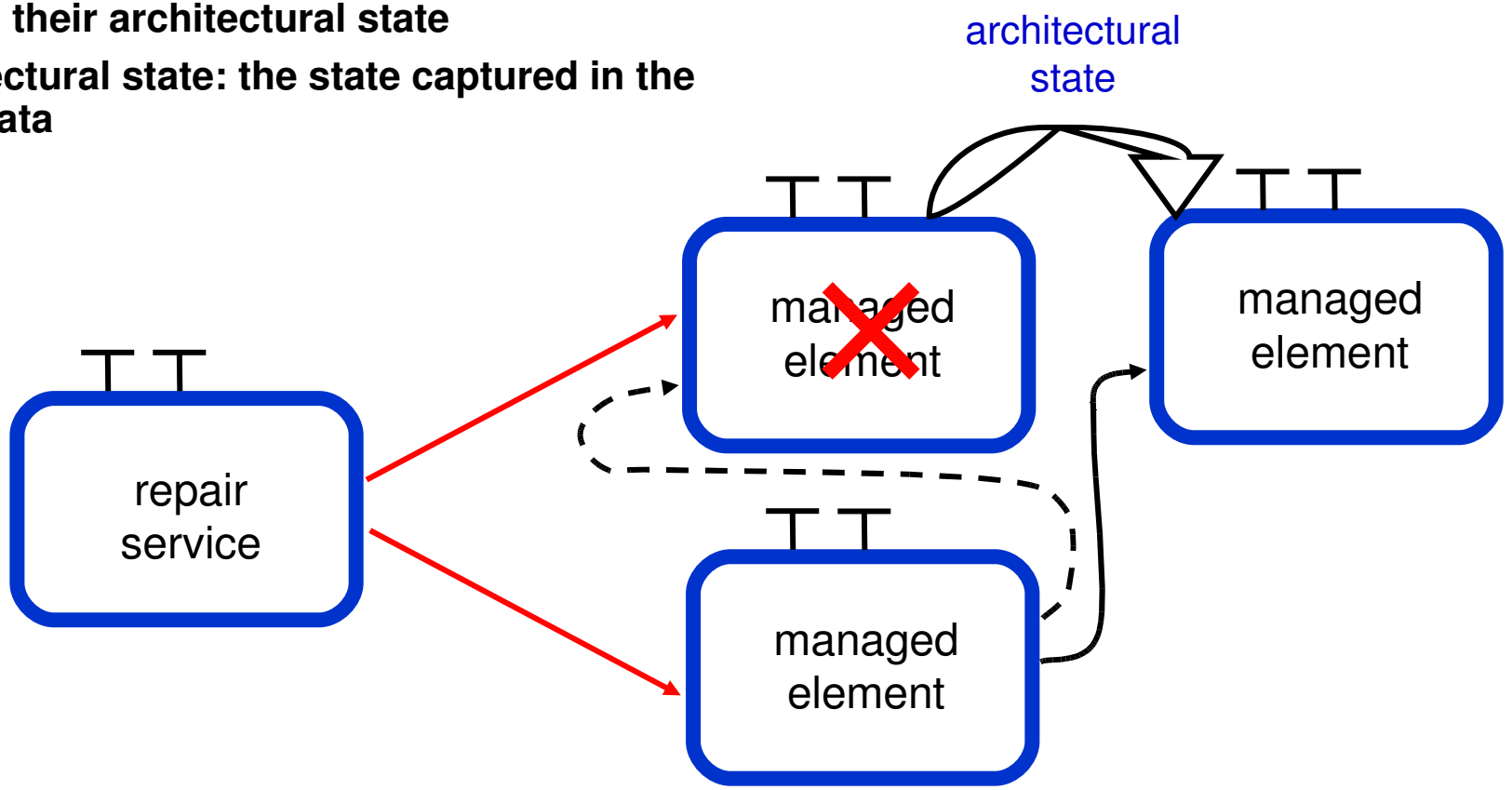
The Jade self-repair service

- **Assumptions**
  - The managed system runs on a cluster of nodes (with a pool of free nodes)
  - In this version, only node failures (fail-stop) are considered
    - Software failures are being investigated
- **Objectives**
  - To provide self-repair for the managed system
  - To provide self-repair for the self-repair service (self-self-repair)



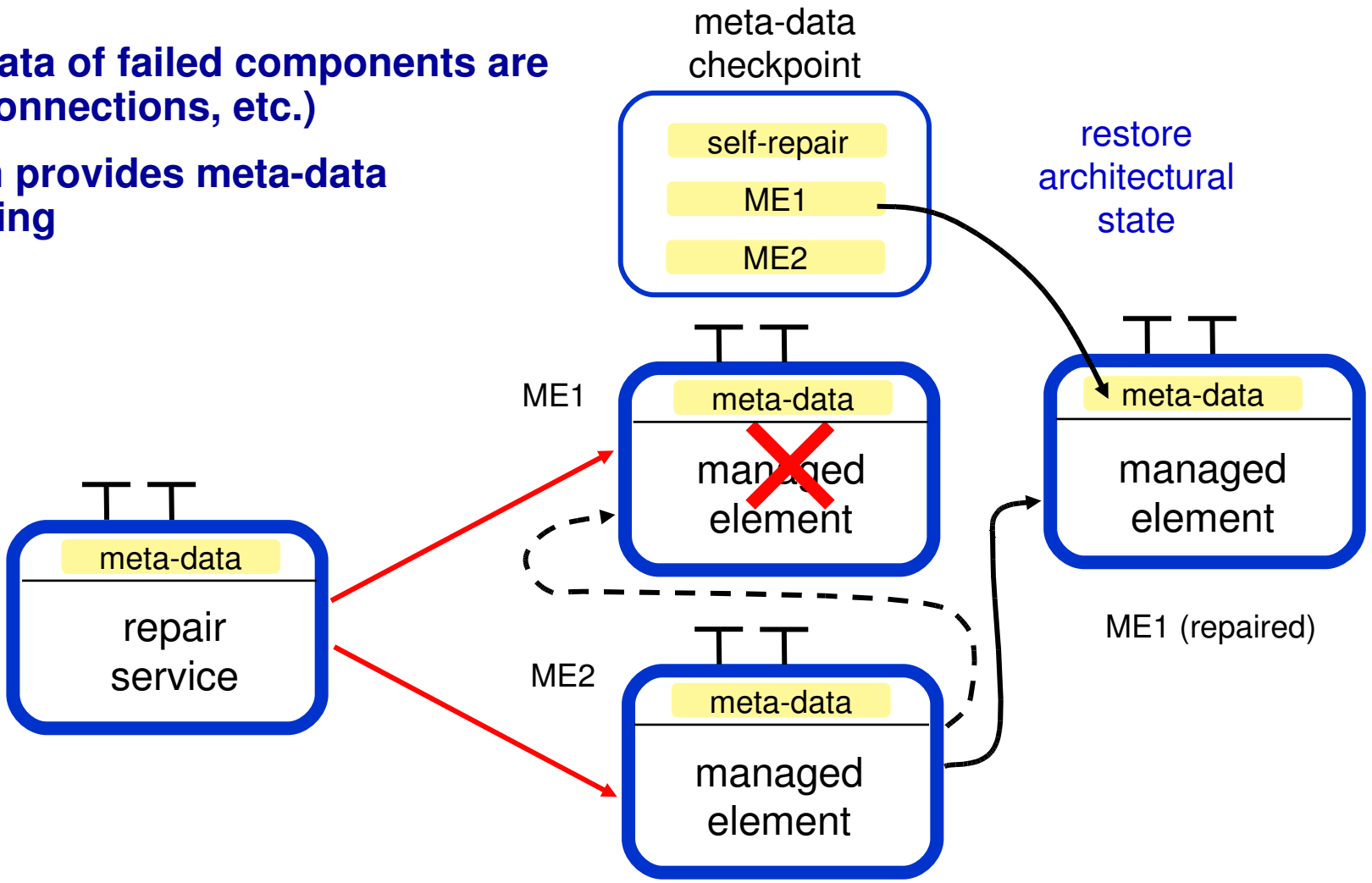
Self-repair principles

- **Repair policy**
  - **Identify failed components and get their architectural state**
  - **Substitute failed components by new ones and restore their architectural state**
  - **Architectural state: the state captured in the meta-data**

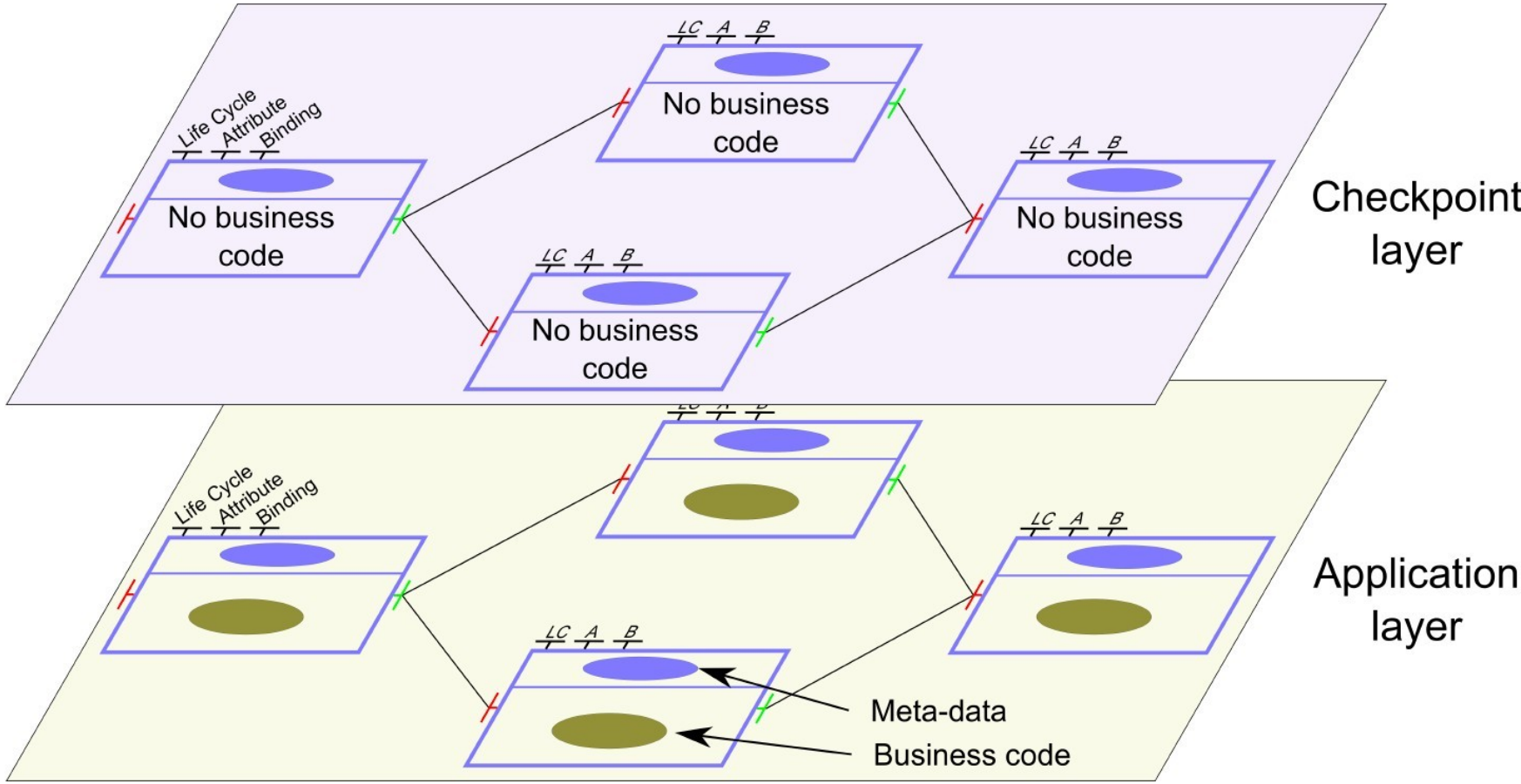


Checkpointing architectural state (1)

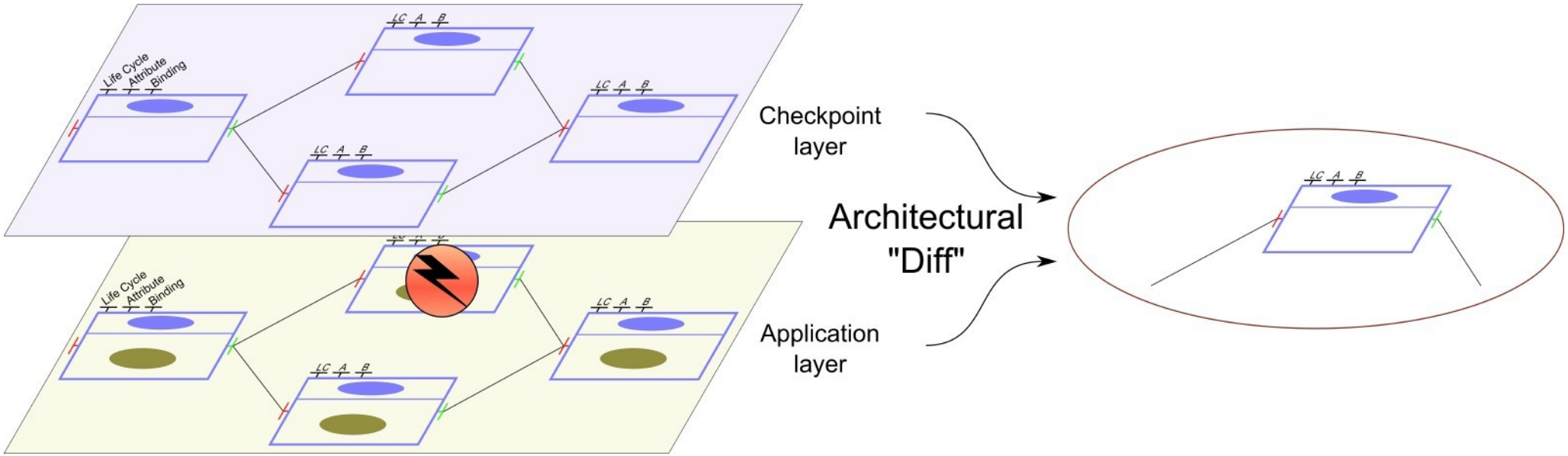
- **The meta-data of failed components are lost (e.g., connections, etc.)**
- **The system provides meta-data checkpointing**



Checkpointing architectural state (2)



Failure analysis



Failed components are identified by comparing the current state of the layer with the checkpointed state

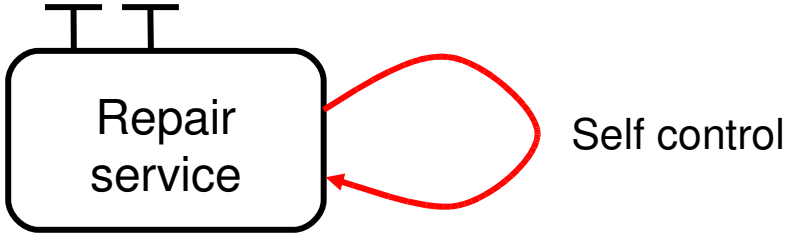
The current state is maintained using usual failure detection techniques (heartbeat)

Making the self-repair system robust (1)

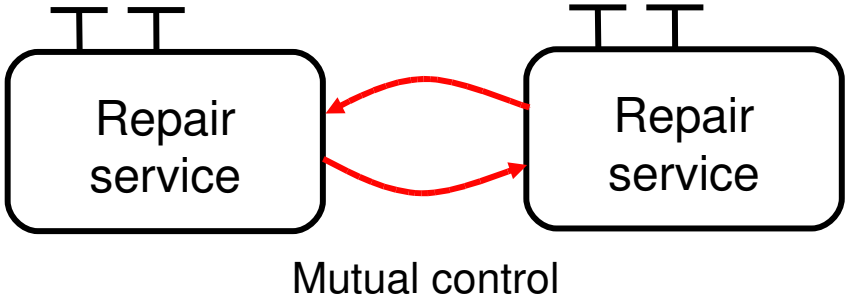
- **Bases of self-repair**
  - Reflective components
  - Architectural state checkpointing
  - Failure detection
- **The self-repair system itself is a single point of failure…**
- **Self-self-repair**
  - The same algorithm is applied recursively
  - This is possible since the self-repair system is structured in reflective components
  - Recursion stops at this level (no self-self-self repair…)

Making the self-repair system robust (2)

- Apply the repair algorithm on the components of self-repair system
- Conceptual view

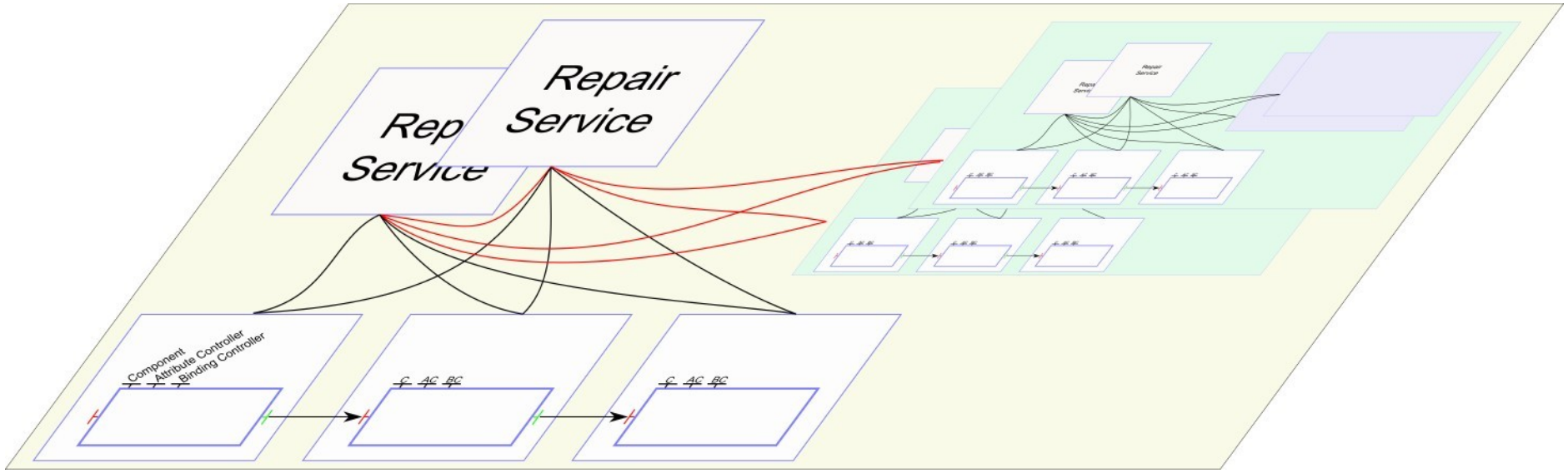


- Implementation view



- Mutual control of replicas
  - Similar to classical process pairs (Tandem, etc.)
  - Each replica works as a component

Putting it all together



The managed application

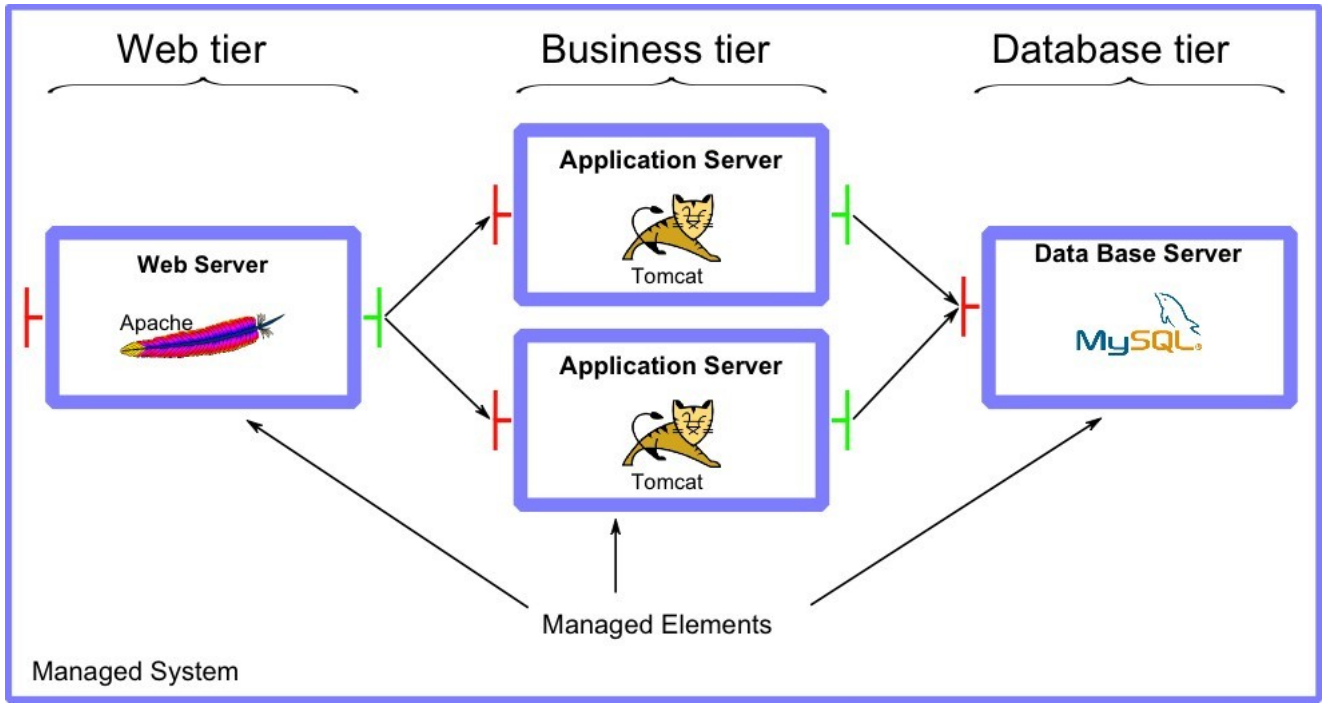
The self-repair service and the checkpoint layer

Self-self repair

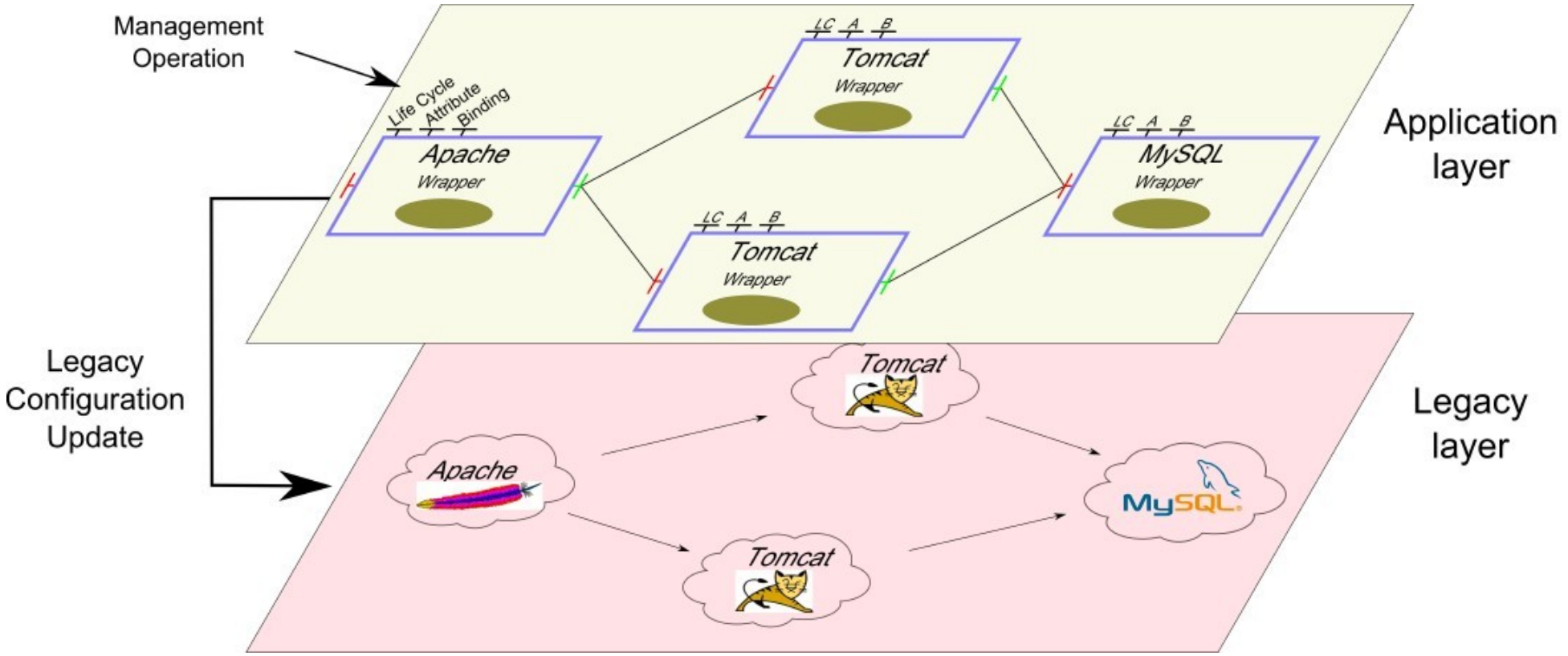
A case study: J2EE application server

Motivations for this example

- A “real life” system with non-trivial architectural complexity
- Combines different legacy middleware technologies
- Used as a test bed for research results

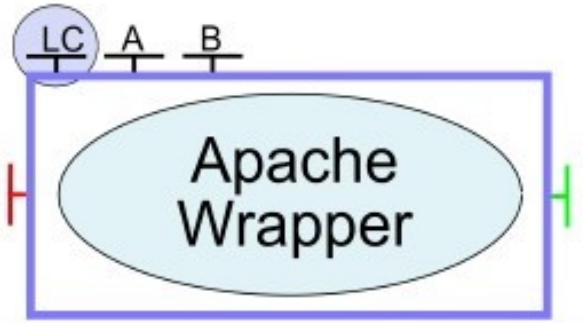


Managing legacy applications (1)



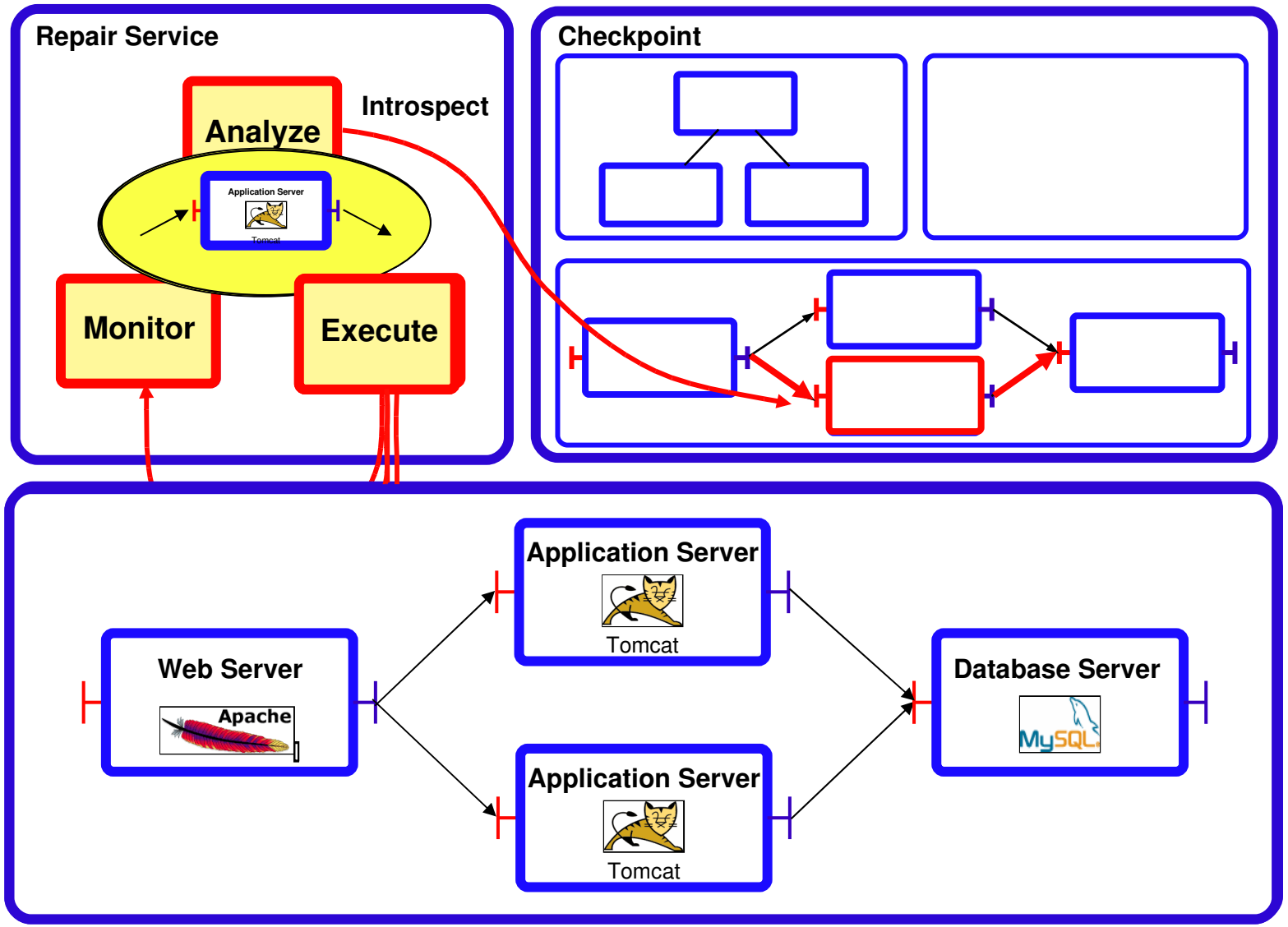
Managing legacy applications (2)

```
public Interface LifeCycleItf {
   /** Starts the component to which this interface belongs. */
   public void startFc() ;

   /** Stops the component to which this interface belongs. */
   public void stopFc() ;
   ...
```



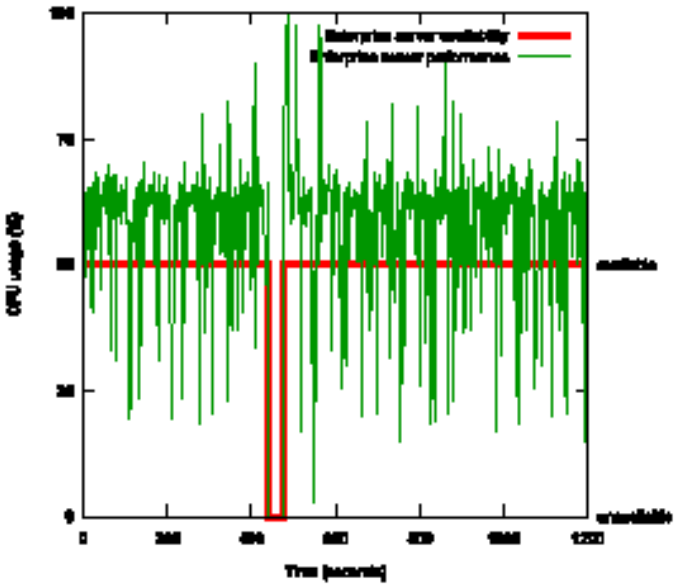

```
Public class ApacheWrapperImpl implements LifeCycleController, ... {

  public void startFc ( ) throws JadeException {
     ShellCommand.syncExec(dirInstall + "/bin/httpd -f " + dirLocal+ "/conf/httpd.conf");
  }

  public void stopFc( ) throws JadeException {
     BufferedReader br = new BufferedReader(new FileReader(dirLocal + "/logs/httpd.pid"));
     String pid = br.readLine();
     ShellCommand.asyncExec( "kill -TERM " + pid );
  }
...
```

The repair algorithm



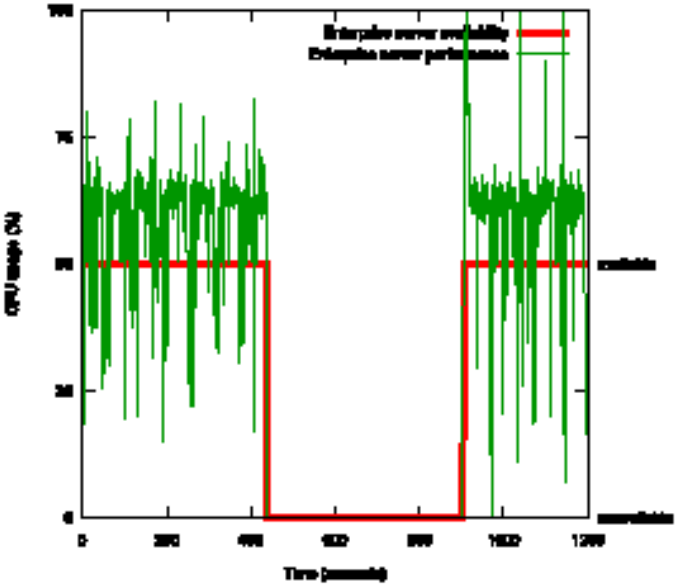
Experimental results

nb. failed req.	1900
MTTR	43 s
Availability	0.96



Automatic recovery

nb. failed req.	18700
MTTR	464 s
Availability	0.72



Recovery managed by a skilled operator

RUBiS workload on a Linux cluster, 1 GB RAM/node, 100Mb Ethernet

	Automatic	Manual
Throughput	12 req./s	12 req./s
Resp. time	89 ms	87 ms
Mem. usage	20,1%	17.5%

Conclusion on Jade

- A case for architecture-base autonomic computing
  - Main construct: reflective components
- Identifying needed properties for self-repair
  - Run time abstractions
    - Attributes, interfaces, lifecycle state, binding, containment
  - Operations
    - Specializable meta-operation (*addSubComponent*, *bind*, etc.)
  - Checkpointing and replication
- Actual applications
  - J2EE server, JMS message server
- Current limitations
  - Only node failures are considered (not software faults)
    - Work in progress…

Some challenges of autonomic computing (1)

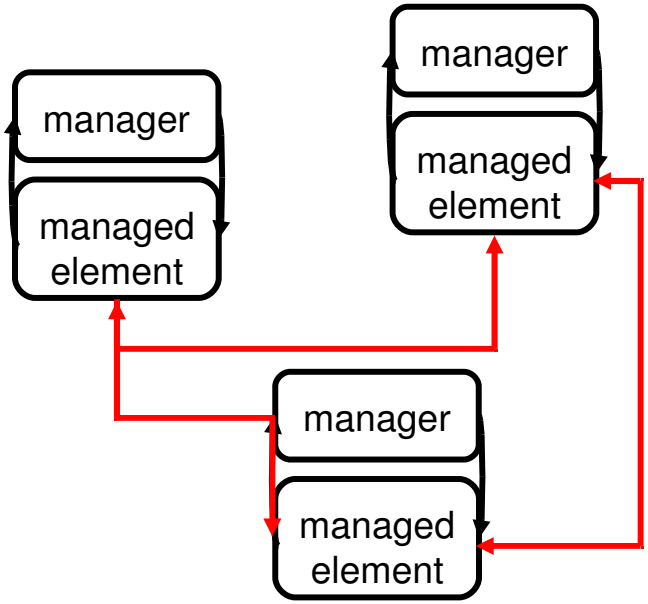
- **Optimization aspects**
  - Models, planning, learning
  - Managing conflicts
  - Scalability
- **Technical aspects**
  - Standards for management interfaces
  - Specific function-related aspects (configuration, repair, etc.)
- **Utility functions**
  - Setting objectives
  - Measuring progress
  - Negotiation
  - Economic analogy
- **Interfacing with humans**
  - Defining and expressing high-level policies
  - Handling emergencies

Some challenges of autonomic computing (2)

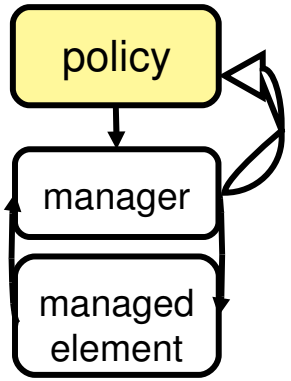
- **Monitoring and analysis**
  - Standards for monitored data
    - Collection
    - Representation
  - Interpreting data
    - Correlation between low-level and high level metrics
  - Locating problems
    - Fine-grain localization of software failures
  - Remedying problems
    - Micro-reboot techniques
    - Learning by fault-injection/repair action
- **Configuration issues**
  - How to define a “correct” configuration?
    - Identifying “good” configurations
    - Inferring constraints

Two (mostly unsolved) problems

- From local to global control
  - Problems
    - Locally optimal ≠ globally optimal
    - Conflict resolution
  - Solutions
    - Super-controller?
    - Collaboration?



- From policy to mechanisms
  - How to express policy?
    - High-level goals
  - How to translate policy into action rules?
  - What about learning?



Towards autonomic computing

- **A recently established research area**
  - New journals, conferences and workshops
    - International Conference on Autonomic Computing (ICAC), IEEE, since 2004
    - ACM Transactions on Autonomous and Adaptive Systems (TAAS), since 2006
    - … many others
- **An important effort in the industry**
  - Autonomic features increasingly present in working applications
- **A number of challenges ahead**